«Книга написана ведущими специалистами в этой области и представляет собой единственное полное изложение предмета».
— Илон Маск,
сооснователь компаний Tesla и SpaceX

Цветное издание

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Я. Гудфеллоу, И. Бенджио, А. Курвилль

Глубокое обучение

Ян Гудфеллоу, Иошуа Бенджио, Аарон Курвилль

Глубокое обучение

Ian Goodfellow, Yoshua Bengio, Aaron Courville

Deep Learning

The MIT Press
Cambridge, Massachusetts
London, England

Ян Гудфеллоу, Иошуа Бенджио, Аарон Курвилль

Глубокое обучение

Второе цветное издание, исправленное

ДМК Пресс
издательство
Москва, 2018

УДК 004.85
ББК 32.971.3
Г93

Гудфеллоу Я., Бенджио И., Курвилль А.
Г93 Глубокое обучение / пер. с анг. А. А. Слинкина. – 2-е изд., испр. – М.: ДМК Пресс, 2018. – 652 с.: цв. ил.

ISBN 978-5-97060-618-6

Глубокое обучение — это вид машинного обучения, наделяющий компьютеры способностью учиться на опыте и понимать мир в терминах иерархии концепций. Книга содержит математические и концептуальные основы линейной алгебры, теории вероятностей и теории информации, численных расчетов и машинного обучения в том объеме, который необходим для понимания материала. Описываются приемы глубокого обучения, применяемые на практике, в том числе глубокие сети прямого распространения, регуляризация, алгоритмы оптимизации, сверточные сети, моделирование последовательностей и др. Рассматриваются такие приложения, как обработка естественных языков, распознавание речи, компьютерное зрение, онлайновые рекомендательные системы, биоинформатика и видеоигры.

Издание предназначено студентам вузов и аспирантам, а также опытным программистам, которые хотели бы применить глубокое обучение в составе своих продуктов или платформ.

УДК 004.85
ББК 32.971.3

ISBN 978-1-491-93799-0 (анг.)
ISBN 978-5-97060-618-6 (рус.)

Содержание

Веб-сайт

www.deeplearning.book

Книгу сопровождает указанный выше сайт, где представлены упражнения, слайды, исправления ошибок и другие материалы, полезные читателям и преподавателям.

Благодарности

Эта книга не состоялась бы, если бы не помощь со стороны многих людей.

Мы благодарны тем, кто откликнулся на наше предложение о написании книги и помог спланировать ее содержание и структуру: Гийом Ален (Guillaume Alain), Кюнгхюн Чо (Kyunghyun Cho), Чаглар Гюльчехре (Çaglar Gülçehre), Дэвид Крюгер (David Krueger), Гуго Ларошель (Hugo Larochelle), Разван Паскану (Razvan Pascanu) и Томас Рохе (Thomas Rohée).

Мы также благодарны всем, кто присылал отзывы о самой книге, иногда даже о нескольких главах: Мартен Абади (Martín Abadi), Гийом Ален, Йон Андруцопулос (Ion Androutsopoulos), Фред Берщ (Fred Bertsch), Олекса Биланюк (Olexa Bilaniuk), Уфук Джан Бичиджи (Ufuk Can Biçici), Матко Босняк (Matko Bošnjak), Джон Буерсма (John Boersma), Грег Брокман (Greg Brockman), Александр де Бребиссон (Alexandre de Brébisson), Пьер Люк Каррье (Pierre Luc Carrier), Саратх Чандар (Sarath Chandar), Павел Чилински (Pawel Chilinski), Марк Дауст (Mark Daoust), Олег Дашевский, Лоран Дин (Laurent Dinh), Стефан Дрезейтль (Stephan Dreseitl), Джим Фан (Jim Fan), Мяо Фан (Miao Fan), Мейре Фортунато (Meire Fortunato), Фредерик Франсис (Frédéric Francis), Нандо де Фрейтас (Nando de Freitas), Чаглар Гюльчехре (Çaglar Gülçehre), Юрген Ван Гаэл (Jurgen Van Gael), Хавьер Алонсо Гарсиа (Javier Alonso García), Джонатан Хант (Jonathan Hunt), Гопи Джейярам (Gopi Jeyaram), Чингиз Кабытаев (Chingiz Kabytayev), Лукаш Кайзер (Lukasz Kaiser), Варун Канаде (Varun Kanade), Азифулла Хан (Asifullah Khan), Акиель Хан (Akiel Khan), Джон Кинг (John King), Дидерик П. Кингма (Diederik P. Kingma), Ян Лекун (Yann LeCun), Рудольф Матей (Rudolf Mathey), Матиас Маттамала (Matías Mattamala), Абхинав Мауриа (Abhinav Maurya), Кэвин Мерфи (Kevin Murphy), Олег Мюрк (Oleg Mürk), Роман Новак (Roman Novak), Огастес К. Одена (Augustus Q. Odena), Симон Павлик (Simon Pavlik), Карл Пичотта (Karl Pichotta), Эдди Пирс (Eddie Pierce), Кари Пулли (Kari Pulli), Руссель Рахман (Roussel Rahman), Тапани Райко (Tapani Raiko), Анурга Ранджан (Anurag Ranjan), Йоханнес Ройт (Johannes Roith), Михаэла Роска (Mihaela Rosca), Халис Сак (Halis Sak), Сезар Салгадо (César Salgado), Григорий Сапунов, Ёсинори Сасаки (Yoshinori Sasaki), Майк Шустер (Mike Schuster), Джулиан Сербан (Julian Serban), Нир Шабат (Nir Shabat), Кен Ширрифф (Ken Shirriff), Андрэ Симпело (Andre Simpelo), Дэвид Слейт (David Slate), Скотт Стэнли (Scott Stanley), Давид Суссилло (David Sussillo), Илья Суцкевер (Ilya Sutskever), Карлес Гелада Саец (Carles Gelada Sáez), Грэхэм Тейлор (Graham Taylor), Валентин Толмер (Valentin Tolmer), Массимильяно Томассоли (Massimiliano Tomassoli), Ан Тран (An Tran), Шубхенду Триведи (Shubhendu Trivedi), Алексей Умнов, Винсет Ванхоуке (Vincent Vanhoucke), Марко Висентини-Скарцанелла (Marco Visentini-Scarzanella), Матрин Вита (Martin Vita), Дэвид Уорд-Фарли (David Warde-Farley), Дастин Уэбб (Dustin Webb), Кэлвин Су (Kelvin Xu), Вэй Сюэ (Wei Xue), Ке Янг (Ke Yang), Ли Яо (Li Yao), Зигмунт Заяц (Zygmunt Zajac) и Озан Чаглаян (Ozan Çaglayan).

Мы признательны и тем, кто делился своим мнением об отдельных главах.

- Обозначения: Чжан Юань Хан (Zhang Yuanhang).
- Глава 1 «Введение»: Юсуф Акгуль (Yusuf Akgul), Себастьен Братьерес (Sebastien Bratieres), Самира Эбрахими (Samira Ebrahimi), Чарли Горичаназ (Charlie

Gorichanaz), Брендан Лоудермилк (Brendan Loudermilk), Эрис Моррис (Eric Morris), Космин Пярвулеску (Cosmin Pârvulescu) и Альфредо Солано (Alfredo Solano).

- Глава 2 «Линейная алгебра»: Амджад Алмахаири (Amjad Almahairi), Никола Баниц (Nikola Banic), Кэвин Беннетт (Kevin Bennett), Филипп Кастонге (Philippe Castonguay), Оскар Чанг (Oscar Chang), Эрик Фослер-Люссье (Eric Fosler-Lussier), Андрей Халявин, Сергей Орешков (Sergey Oreshkov), Иштван Петрас (István Petrás), Дэннис Прэнгл (Dennis Prangle), Томас Рохе (Thomas Rohée), Гитанджали Гулве Сехгал (Gitanjali Gulve Sehgal), Колби Толанд (Colby Toland), Алессандро Витале (Alessandro Vitale) и Боб Уэлланд (Bob Welland).
- Глава 3 «Теория вероятностей и теория информации»: Джон Филипп Андерсон (John Philip Anderson), Кай Арулкумаран (Kai Arulkumaran), Венсан Дюмуле (Vincent Dumoulin), Руй Фа (Rui Fa), Стефан Гоус (Stephan Gouws), Артем Оботуров, Антти Расмус (Antti Rasmus), Алексей Сурков и Фолькер Тресп (Volker Tresp).
- Глава 4 «Численные методы»: Тран Лам Аниан Фишер (Tran Lam AnIan Fischer) и Ху Ю Хуан (Hu Yuhuang).
- Глава 5 «Основы машинного обучения»: Дзмитри Бахданау (Dzmitry Bahdanau), Жюстен Доманге (Justin Domingue), Нихил Гарг (Nikhil Garg), Макото Оцука (Makoto Otsuka), Боб Пепин (Bob Pepin), Филип Попьен (Philip Popien), Бхарат Прабхакар (Bharat Prabhakar), Эммануэль Райнер (Emmanuel Rayner), Питер Шепард (Peter Shepard), Ки-Бонг Сонг (Kee-Bong Song), Чжен Сун (Zheng Sun) и Энди Ву (Andy Wu).
- Глава 6 «Глубокие сети прямого распространения»: Уриэль Бердуго (Uriel Berdugo), Фабрицио Боттарель (Fabrizio Bottarel), Элизабет Бэрл (Elizabeth Burl), Ишан Дуругкар (Ishan Durugkar), Джефф Хлыва (Jeff Hlywa), Ёнг Вук Ким (Jong Wook Kim), Давид Крюгер (David Krueger), Адития Кумар Прахарадж (Aditya Kumar Praharaj) и Стэн Сутла (Sten Sootla).
- Глава 7 «Регуляризация в глубоком обучении»: Мортен Колбэк (Morten Kolbæk), Читыдж Лауриа (Kshitij Lauria), Инкю Ли (Inkyu Lee), Сунил Мохан (Sunil Mohan), Хай Пхонг Пхан (Hai Phong Phan) и Джошуа Сэлисбэри (Joshua Salisbury).
- Глава 8 «Оптимизация обучения глубоких моделей»: Марсель Аккерман (Marcel Ackermann), Питер Армитейдж (Peter Armitage), Роуэл Атиенза (Rowel Atienza), Эндрю Брок (Andrew Brock), Теган Махарадж (Tegan Maharaj), Джеймс Мартенс (James Martens), Мостафа Натегх (Mostafa Nategh), Кашиф Расул (Kashif Rasul), Клаус Штробль (Klaus Strobl) и Никола Тэрнер (Nicholas Turner).
- Глава 9 «Сверточные сети»: Мартен Аржовски (Martín Arjovsky), Евгений Бревдо (Eugene Brevdo), Константин Дивилов, Эрик Йенсен (Eric Jensen), Мехди Мирза (Mehdi Mirza), Алекс Пайно (Alex Paino), Марджори Сэйер (Marjorie Sayer), Райан Стаут (Ryan Stout) и Вентао Ву (Wentao Wu).
- Глава 10 «Моделирование последовательностей: рекуррентные и рекурсивные сети»: Гёкчен Ераслан (Gökçen Eraslan), Стивен Хиксон (Steven Hickson), Разван Паскану (Razvan Pascanu), Лоренхо фон Риттер (Lorenzo von Ritter), Руй Родригес (Rui Rodrigues), Дмитрий Сердюк, Донгуй Ши (Dongyu Shi) и Кай Ю Яан (Kaiyu Yang).

- Глава 11 «Практическая методология»: Даниэль Бекштейн (Daniel Beckstein).
- Глава 12 «Приложения»: Джордж Дал (George Dahl), Владимир Некрасов (Vladimir Nekrasov) и Рибана Рошер (Ribana Roscher).
- Глава 13 «Линейные факторные модели»: Джейнт Кушик (Jayanth Koushik).
- Глава 15 «Обучение представлений»: Кунал Гхош (Kunal Ghosh).
- Глава 16 «Структурные вероятностные модели в глубоком обучении»: Минь Ле (Minh Lê) и Антон Варфолом (Anton Varfolom).
- Глава 18 «Преодоление трудностей, связанных со статической суммой»: Сэм Боумен (Sam Bowman).
- Глава 19 «Приближенный вывод»: Ю Цзя Бао (Yujia Bao).
- Глава 20 «Глубокие порождающие модели»: Николас Чападос (Nicolas Chapados), Даниэль Галвес (Daniel Galvez), Вэн Мин Ма (Wenming Ma), Фади Медхат (Fady Medhat), Шакри Мохамед (Shakir Mohamed) и Грегуар Монтавон (Grégoire Montavon).
- Библиография: Лукас Михельбахер (Lukas Michelbacher) и Лесли Н. Смит (Leslie N. Smith).

Мы также благодарим авторов, давших нам разрешение использовать в тексте изображения, рисунки и данные из их публикаций. Источники указываются в подрисуночных подписях.

Мы благодарны Лю Вану (Lu Wang), написавшему программу pdf2htmlEX, которой мы пользовались для подготовки варианта книги для веба, а также за помощь в улучшении качества получившегося HTML-кода.

Спасибо супруге Яна Даниэле Флори Гудфеллоу за терпеливую поддержку Яна на всем протяжении работы над книгой и за помощь в выверке текста.

Спасибо команде Google Brain за создание атмосферы интеллектуального общения, позволившей Яну уделять много времени работе над книгой и получать отзывы и наставления от коллег. Особенно хотим поблагодарить бывшего начальника Яна, Грега Коррадо (Greg Corrado), и его нынешнего начальника, Сами Бенджио (Samy Bengio), за поддержку этого проекта. Наконец, мы благодарны Джеффри Хинтону (Geoffrey Hinton), который подбадривал нас, когда было трудно.

Обозначения

В этом разделе приведен краткий перечень обозначений, используемых в книге. Большая часть соответствующего математического аппарата описана в главах 2–4.

Числа и массивы

a	скаляр (целый или вещественный)
$\boldsymbol{a}$	вектор
$\boldsymbol{A}$	матрица
$\mathbf{A}$	тензор
$\boldsymbol{I}_n$	единичная матрица с n строками и n столбцами
$\boldsymbol{I}$	единичная матрица, размер которой определяется контекстом
$\boldsymbol{e}^{(i)}$	стандартный базисный вектор [0, ..., 0, 1, 0..., 0], содержащий 1 в i-й позиции
diag($\boldsymbol{a}$)	квадратная диагональная матрица, на диагонали которой находятся элементы вектора $\boldsymbol{a}$
a	случайная скалярная величина
$\mathbf{a}$	случайный вектор
$\mathbf{A}$	случайная матрица

Множества и графы

$\mathbb{A}$	множество
$\mathbb{R}$	множество вещественных чисел
$\{0, 1\}$	множество из двух элементов: 0 и 1
$\{0, 1, ..., n\}$	множество целых чисел от 0 до n включительно
$[a, b]$	замкнутый интервал вещественной прямой от a до b, включающий границы
$(a, b]$	интервал вещественной прямой, не включающий a, но включающий b
$\mathbb{A}\setminus\mathbb{B}$	разность множеств, т. е. множество, содержащее все элементы $\mathbb{A}$, не являющиеся элементами $\mathbb{B}$
$\mathcal{G}$	граф
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	родители x_i в $\mathcal{G}$

Индексирование

a_i	i-й элемент вектора $\boldsymbol{a}$, индексирование начинается с 1
a_{-i}	все элементы вектора $\boldsymbol{a}$, кроме i-го
$A_{i,j}$	элемент матрицы $\boldsymbol{A}$ в позиции (i, j)
$\boldsymbol{A}_{i,:}$	i-я строка матрицы $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	i-й столбец матрицы $\boldsymbol{A}$
$\mathbf{A}_{i,j,k}$	элемент трехмерного тензора $\mathbf{A}$ в позиции (i, j, k)
$\mathbf{A}_{:,:,i}$	двумерная срезка трехмерного тензора $\mathbf{A}$
a_i	i-й элемент случайного вектора $\mathbf{a}$

Операции линейной алгебры

$\boldsymbol{A}^\top$	матрица, транспонированная к $\boldsymbol{A}$
$\boldsymbol{A}^+$	псевдообратная матрица Мура-Пенроуза

$\boldsymbol{A} \odot \boldsymbol{B}$	поэлементное произведение матриц $\boldsymbol{A}$ и $\boldsymbol{B}$ (произведение Адамара)
$\det(\boldsymbol{A})$	определитель $\boldsymbol{A}$

Математический анализ

$\frac{dy}{dx}$	производная y по x
$\frac{\partial y}{\partial x}$	частная производная y по x
$\nabla_{\boldsymbol{x}} y$	градиент y по $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	матрица производных y относительно $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	тензор производных y относительно $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	матрица Якоби $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ функции $f\colon \mathbb{R}^n \to \mathbb{R}^m$
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ или $\boldsymbol{H}(f)(\boldsymbol{x})$	гессиан функции f в точке $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	определенный интеграл по всей области определения $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	определенный интеграл по множеству $\mathbb{S}$

Теория вероятностей и теория информации

$\mathrm{a} \perp \mathrm{b}$	случайные величины a и b независимы
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	они условно независимы при условии c
$P(\mathrm{a})$	распределение вероятности дискретной случайной величины
$p(\mathrm{a})$	распределение вероятности непрерывной случайной величины или величины, тип которой не задан
$\mathrm{a} \sim P$	случайная величина a имеет распределение P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ или $\mathbb{E} f(x)$	математическое ожидание $f(x)$ при заданном распределении $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	дисперсия $f(x)$ при заданном распределении $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	ковариация $f(x)$ и $g(x)$ при заданном распределении $P(\mathrm{x})$
$H(\mathrm{x})$	энтропия Шеннона случайной величины x
$D_{\mathrm{KL}}(P \Vert Q)$	расхождение Кульбака–Лейблера между P и Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	нормальное распределение случайной величины $\boldsymbol{x}$ со средним $\boldsymbol{\mu}$ и ковариацией $\boldsymbol{\Sigma}$

Функции

$f\colon \mathbb{A} \to \mathbb{B}$	функция f с областью определения $\mathbb{A}$ и областью значений $\mathbb{B}$
$f \circ g$	композиция функций f и g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	функция от $\boldsymbol{x}$, параметризованная $\boldsymbol{\theta}$ (иногда, чтобы не утяжелять формулы, мы пишем $f(\boldsymbol{x})$, опуская аргумент $\boldsymbol{\theta}$)
$\log x$	натуральный логарифм x
$\sigma(x)$	логистическая сигмоида, $1 / (1 + \exp(-x))$
$\zeta(x)$	функция $\log(1 + \exp(x))$
$\Vert \boldsymbol{x} \Vert_p$	норма L^p вектора $\boldsymbol{x}$
$\Vert \boldsymbol{x} \Vert$	норма L^2 вектора $\boldsymbol{x}$
x^+	положительная часть x, т. е. $\max(0, x)$
$\mathbf{1}_{\mathrm{condition}}$	1, если условие *condition* истинно, иначе 0

Иногда мы применяем функцию f со скалярным аргументом к вектору, матрице или тензору: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$ или $f(\mathbf{X})$. Это означает, что функция f применяется к каждому элементу массива. Например, запись $\mathbf{C} = \sigma(\mathbf{X})$ означает, что $C_{i,j,k} = \sigma(X_{i,j,k})$ для всех i, j, k.

Наборы данных и распределения

p_{data}	распределение, порождающее данные
$\hat{p}_{\text{data}}$	эмпирическое распределение, определенное обучающим набором
$\mathbb{X}$	обучающий набор примеров
$\boldsymbol{x}^{(i)}$	i-й пример из входного набора данных
$y^{(i)}$ или $\boldsymbol{y}^{(i)}$	метка, ассоциированная с $\boldsymbol{x}^{(i)}$ при обучении с учителем
$\boldsymbol{X}$	матрица $m \times n$, в строке $\boldsymbol{X}_{i,:}$ которой находится входной пример $\boldsymbol{x}^{(i)}$

Глава 1

Введение

Изобретатели давно мечтали создать думающую машину. Эти мечты восходят еще к Древней Греции. Персонажей мифов – Пигмалиона, Дедала, Гефеста – можно было бы назвать легендарными изобретателями, а их творения – Галатею, Талоса и Пандору – искусственной жизнью (Ovid and Martin, 2004; Sparkes, 1996; Tandy, 1997).

Впервые задумавшись о программируемых вычислительных машинах, человек задался вопросом, смогут ли они стать разумными, – за сотню с лишним лет до построения компьютера (Lovelace, 1842). Сегодня **искусственный интеллект** (ИИ) – бурно развивающаяся дисциплина, имеющая многочисленные приложения. Мы хотим иметь интеллектуальные программы, которые могли бы автоматизировать рутинный труд, понимали речь и изображения, ставили медицинские диагнозы и поддерживали научные исследования.

Когда наука об искусственном интеллекте только зарождалась, были быстро исследованы и решены некоторые задачи, трудные для человека, но относительно простые для компьютеров – описываемые с помощью списка формальных математических правил. Настоящим испытанием для искусственного интеллекта стали задачи, которые легко решаются человеком, но с трудом поддаются формализации, – задачи, которые мы решаем интуитивно, как бы автоматически: распознавание устной речи или лиц на картинке.

Эта книга посвящена решению таких интуитивных задач. Цель заключается в том, чтобы компьютер мог учиться на опыте и понимать мир в терминах иерархии понятий, каждое из которых определено через более простые понятия. Благодаря приобретению знаний опытным путем этот подход позволяет исключить этап формального описания человеком всех необходимых компьютеру знаний. Иерархическая организация дает компьютеру возможность учиться более сложным понятиям путем построения их из более простых. Граф, описывающий эту иерархию, будет глубоким – содержащим много уровней. Поэтому такой подход к ИИ называется **глубоким обучением**.

Ранние успехи ИИ в большинстве своем были достигнуты в относительно стерильной формальной среде, где от компьютера не требовались обширные знания о мире. Взять, к примеру, созданную IBM шахматную программу Deep Blue, которая в 1997 году обыграла чемпиона мира Гарри Каспарова (Hsu, 2002). Шахматы – это очень простой мир, состоящий всего из 64 клеток и 32 фигур, которые могут ходить лишь строго определенным образом. Разработка успешной стратегии игры в шахматы – огромное достижение, но трудность задачи – не в описании множества фигур и допустимых ходов на языке, понятном компьютеру. Для полного описания игры достаточно очень короткого списка формальных правил, который заранее составляется программистом.

Забавно, что абстрактные, формально поставленные задачи, требующие значительных умственных усилий от человека, для компьютера как раз наиболее просты. Компьютеры давно уже способны обыграть в шахматы сильнейших гроссмейстеров, но лишь в последние годы стали сопоставимы с человеком в части распознавания объектов или речи. В повседневной жизни человеку необходим гигантский объем знаний о мире. Знания эти субъективны и представлены на интуитивном уровне, поэтому выразить их формально затруднительно. Но чтобы вести себя «разумно», компьютерам нужны такие же знания. Одна из основных проблем искусственного интеллекта – как заложить эти неформальные знания в компьютер.

Авторы нескольких проектов в области ИИ пытались представить знания о мире с помощью формальных языков. Компьютер может автоматически рассуждать о предложениях на таком языке, применяя правила логического вывода. В основе таких подходов лежит **база знаний**. Ни один из этих проектов не привел к существенному успеху. Одним из самых известных был проект Cyc (Lenat and Guha, 1989) – машина логического вывода и база утверждений на языке CycL. За ввод утверждений отвечал штат учителей-людей. Процесс оказывается крайне громоздким. Люди изо всех сил пытаются придумать формальные правила, достаточно сложные для точного описания мира. Например, Cyc не сумел понять рассказ о человеке по имени Фред, который бреется по утрам (Linde, 1992). Его машина вывода обнаружила в рассказе противоречие: он знал, что в людях нет электрических деталей, но, поскольку Фред держал электрическую бритву, система решила, что объект «БреющийсяФред» содержит электрические детали. И задала вопрос: является ли Фред по-прежнему человеком, когда бреется.

Трудности, с которыми сталкиваются системы, опирающиеся на «зашитые в код» знания, наводят на мысль, что система с искусственным интеллектом должна уметь самостоятельно накапливать знания, отыскивая закономерности в исходных данных. Это умение называется **машинным обучением**. С появлением машинного обучения перед компьютерами открылась возможность подступиться к задачам, требующим знаний о реальном мире, и принимать решения, кажущиеся субъективными. Простой алгоритм машинного обучения – **логистическая регрессия** – может решить, следует ли рекомендовать кесарево сечение (Mor-Yosef et al., 1990). Другой простой алгоритм – **наивный байесовский классификатор** – умеет отделять нормальную электронную почту от спама.

Качество этих простых алгоритмов сильно зависит от представления исходных данных. Так, система ИИ, выдающая рекомендации о показанности кесарева сечения, не осматривает пациента. Вместо этого врач сообщает системе относящуюся к делу информацию, например о наличии или отсутствии рубца на матке. Каждый отдельный элемент информации, включаемый в представление о пациенте, называется **признаком**. Алгоритм логистической регрессии анализирует, как признаки пациента коррелируют с различными результатами. Но он не может никаким образом повлиять на определение признаков. Если алгоритму предложить снимок МРТ, а не формализованные врачом сведения, то он не сможет выдать полезную рекомендацию. Отдельные пиксели снимка практически не коррелированы с осложнениями, которые могут возникнуть во время родов.

Эта зависимость от представления является общим явлением, проявляющимся как в информатике, так и в повседневной жизни. Если говорить об информатике, то такие операции, как поиск в коллекции данных, будут производиться многократно

быстрее, если коллекция структурирована и подходящим образом индексирована. Люди же легко выполняют арифметические операции с числами, записанными арабскими цифрами, но тратят куда больше времени, если используются римские цифры. Неудивительно, что выбор представления оказывает огромное влияние на качество и производительность алгоритмов машинного обучения. На рис. 1.1 приведен простой наглядный пример.

Многие задачи ИИ можно решить, если правильно подобрать признаки, а затем предъявить их алгоритму машинного обучения. Например, в задаче идентификации говорящего по звукам речи полезным признаком является речевой тракт. Он позволяет с большой точностью определить, кто говорит: мужчина, женщина или ребенок.

Но во многих задачах нелегко понять, какие признаки выделять. Допустим, к примеру, что мы пишем программу обнаружения автомобилей на фотографиях. Мы знаем, что у автомобилей есть колеса, поэтому могли бы счесть присутствие колеса признаком. К сожалению, на уровне пикселей трудно описать, как выглядит колесо. Колесо имеет простую геометрическую форму, но распознавание его изображения может быть осложнено отбрасыванием теней, блеском солнца на металлических деталях автомобиля, наличием щитка, защищающего колесо от грязи, или объектов на переднем плане, частично загораживающих колесо, и т. д.

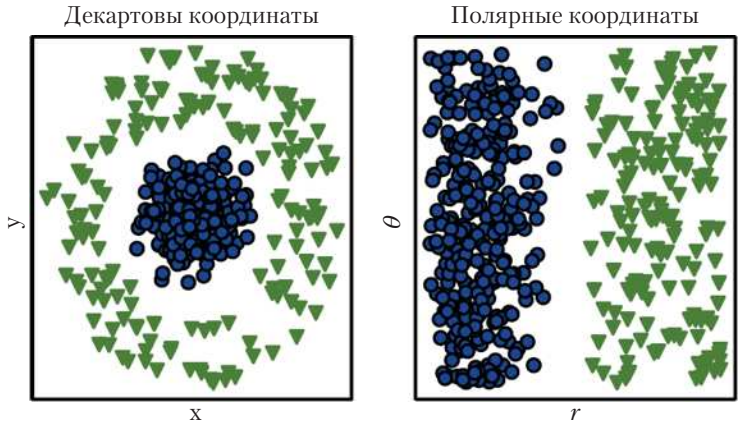

Рис. 1.1 ❖ Пример различных представлений: предположим, что требуется разделить две категории данных, проведя прямую на диаграмме рассеяния. На левом рисунке данные представлены в декартовых координатах, и задача неразрешима. На правом рисунке те же данные представлены в полярных координатах и разделяются вертикальной прямой (рисунок подготовлен совместно с Дэвидом Уорд-Фарли)

Одно из решений этой проблемы – воспользоваться машинным обучением не только для того, чтобы найти отображение представления на результат, но и чтобы определить само представление. Такой подход называется **обучением представлений**. На представлениях, полученных в ходе обучения, часто удается добиться гораздо более высокого качества, чем на представлениях, созданных вручную. К тому же это позволяет системам ИИ быстро адаптироваться к новым задачам при минималь-

ном вмешательстве человека. Для простой задачи алгоритм обучения представлений может найти хороший набор признаков за несколько минут, для сложных – за время от нескольких часов до нескольких месяцев. Проектирование признаков вручную для сложной задачи требует много времени и труда, на это могут уйти десятилетия работы всего сообщества исследователей.

Квинтэссенцией алгоритма обучения представлений является **автокодировщик**. Это комбинация функции **кодирования**, которая преобразует входные данные в другое представление, и функции **декодирования**, которая преобразует новое представление в исходный формат. Обучение автокодировщиков устроено так, чтобы при кодировании и обратном декодировании сохранялось максимально много информации, но чтобы при этом новое представление обладало различными полезными свойствами. Различные автокодировщики ориентированы на получение различных свойств.

При проектировании признаков или алгоритмов обучения признаков нашей целью обычно является выделение **факторов вариативности**, которые объясняют наблюдаемые данные. В этом контексте слово «фактор» означает просто источник влияния, а не «сомножитель». Зачастую факторы – это величины, не наблюдаемые непосредственно. Это могут быть ненаблюдаемые объекты или силы в физическом мире, оказывающие влияние на наблюдаемые величины. Это могут быть также умозрительные конструкции, дающие полезные упрощающие объяснения или логически выведенные причины наблюдаемых данных. Их можно представлять себе как концепции или абстракции, помогающие извлечь смысл из данных, характеризуемых высокой вариативностью. В случае анализа записи речи к факторам вариативности относятся возраст и пол говорящего, акцент и произносимые слова. В случае анализа изображения автомобиля факторами вариативности являются положение машины, ее цвет, а также высота солнца над горизонтом и его яркость.

Источник трудностей в целом ряде практических приложений искусственного интеллекта – тот факт, что многие факторы вариативности оказывают влияние абсолютно на все данные, доступные нашему наблюдению. Отдельные пиксели изображения красного автомобиля ночью могут быть очень близки к черному цвету. Форма силуэта автомобиля зависит от угла зрения. В большинстве приложений требуется *разделить* факторы вариативности и отбросить те, что нам не интересны.

Разумеется, может оказаться очень трудно выделить такие высокоуровневые абстрактные признаки из исходных данных. Многие факторы вариативности, к примеру акцент говорящего, можно идентифицировать, только если наличествует очень глубокое, приближающееся к человеческому понимание природы данных. Но раз получить представление почти так же трудно, как решить исходную задачу, то, на первый взгляд, обучение представлений ничем не поможет.

Глубокое обучение решает эту центральную проблему обучения представлений, вводя представления, выражаемые в терминах других, более простых представлений. Глубокое обучение позволяет компьютеру строить сложные концепции из более простых. На рис. 1.2 показано, как в системе глубокого обучения можно представить концепцию изображения человека в виде комбинации более простых концепций – углов и контуров, – которые, в свою очередь, определены в терминах границ.

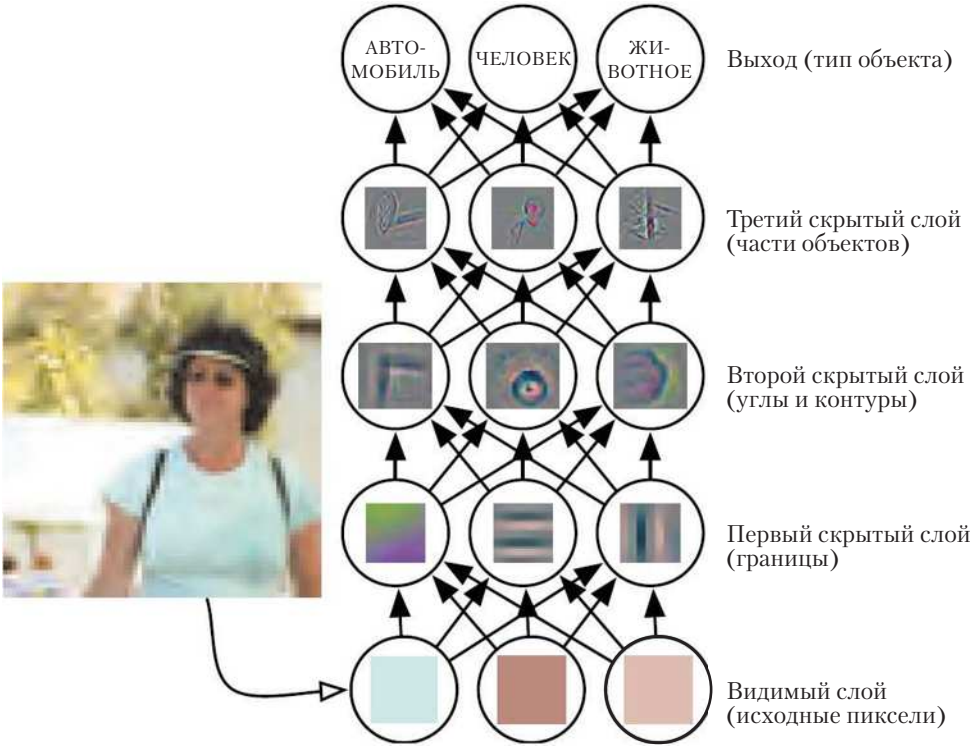

Рис. 1.2 ❖ Иллюстрация модели глубокого обучения. Компьютеру трудно понять смысл исходных данных, полученных от сенсоров, таких, например, как изображение, представленное в виде набора значений пикселей. Функция, отображающая множество пикселей в распознанный объект, очень сложна. Если подходить к задаче вычисления или обучения этого отображения в лоб, то она выглядит безнадежной. Глубокое обучение разрешает эту проблему, разбивая искомое сложное отображение на ряд более простых вложенных, каждое из которых описывается отдельным слоем модели. Входные данные представлены видимым слоем, он называется так, потому что содержит переменные, доступные наблюдению. За ним идет ряд скрытых слоев, которые извлекают из изображения все более и более абстрактные признаки. Слово «скрытый» означает, что значения, вырабатываемые этими слоями, не присутствуют в данных; сама модель должна определить, какие концепции полезны для объяснения связей в наблюдаемых данных. На рисунке показаны признаки, представленные каждым скрытым слоем. Зная исходные пиксели, первый слой легко может найти границы, для этого нужно лишь сравнить яркости соседних пикселей. Имея описание границ, выработанное первым скрытым слоем, второй скрытый слой находит углы и контуры в виде наборов границ. По этому описанию третий скрытый слой может распознать части конкретных объектов, представленные совокупностями контуров и углов определенного вида. Наконец, по описанию изображения в терминах частей объектов можно распознать сами объекты. Изображения взяты из работы Zeiler and Fergus (2014) с разрешения авторов

Типичным примером модели глубокого обучения является глубокая сеть прямого распространения, или **многослойный перцептрон** (МСП). Многослойный перцептрон – это просто математическая функция, отображающая множество входных значений на множество выходных. Эта функция является композицией нескольких более простых функций. Каждое применение одной математической функции можно рассматривать как новое представление входных данных.

Идея нахождения подходящего представления данных путем обучения – это лишь один взгляд на глубокое обучение. Другой взгляд состоит в том, что глубина позволяет обучать многошаговую компьютерную программу. Каждый слой представления можно мыслить себе как состояние памяти компьютера после параллельного выполнения очередного набора инструкций. Чем больше глубина сети, тем больше инструкций она может выполнить последовательно. Последовательное выполнение инструкций расширяет возможности, поскольку более поздние инструкции могут обращаться к результатам выполнения предыдущих.

При таком взгляде на глубокое обучение не всякая информация, используемая для активации слоев, обязательно кодирует факторы вариативности, объясняющие входные данные. В представлении хранится также вспомогательная информация о состоянии, помогающая выполнить программу, способную извлекать смысл из данных. Эту информацию можно уподобить счетчику или указателю в традиционной компьютерной программе. Она не имеет никакого отношения к содержанию входных данных, но помогает модели в организации их обработки.

Есть два основных способа измерить глубину модели. Первый оценивает архитектуру на основе числа последовательных инструкций, которые необходимо выполнить. Можно считать, что это длина самого длинного пути в графе, описывающем вычисление каждого выхода модели по ее входам. Как у двух эквивалентных компьютерных программ могут быть разные длины пути в зависимости от языка, на котором они написаны, так и одна и та же функциональность может быть изображена графами с разной длиной пути в зависимости от того, какие функции допускаются в качестве шагов. На рис. 1.3 показано, как выбор языка может дать разные результаты измерений для одной и той же архитектуры.

При другом подходе, используемом в глубоких вероятностных моделях, глубиной модели считается не глубина графа вычислений, а глубина графа, описывающего связи концепций. В этом случае граф вычислений, выполняемых для вычисления представления каждой концепции, может быть гораздо глубже, чем граф самих концепций. Связано это с тем, что понятие системы о простых концепциях можно уточнять, располагая информацией о более сложных. Например, система ИИ, наблюдающая изображение лица, на котором один глаз находится в тени, первоначально может распознать только один глаз. Но, обнаружив присутствие лица, система может заключить, что должен быть и второй глаз. В таком случае граф концепций содержит только два слоя – для глаз и для лиц, тогда как граф вычислений содержит $2n$ слоев, если мы n раз уточняем оценку каждой концепции при известной информации о второй.

Поскольку не всегда ясно, какой из двух подходов – глубина графа вычислений или глубина графа вероятностной модели – более релевантен, и поскольку разные

люди по-разному выбирают наборы примитивных элементов, из которых строятся графы, не существует единственно правильного значения глубины архитектуры, как не существует единственно правильной длины компьютерной программы. И нет общего мнения о том, какой должна быть глубина, чтобы модель можно было считать «глубокой». Однако можно все-таки сказать, что глубокое обучение – это наука о моделях, в которых уровень композиции обученных функций или обученных концепций выше, чем в традиционном машинном обучении.

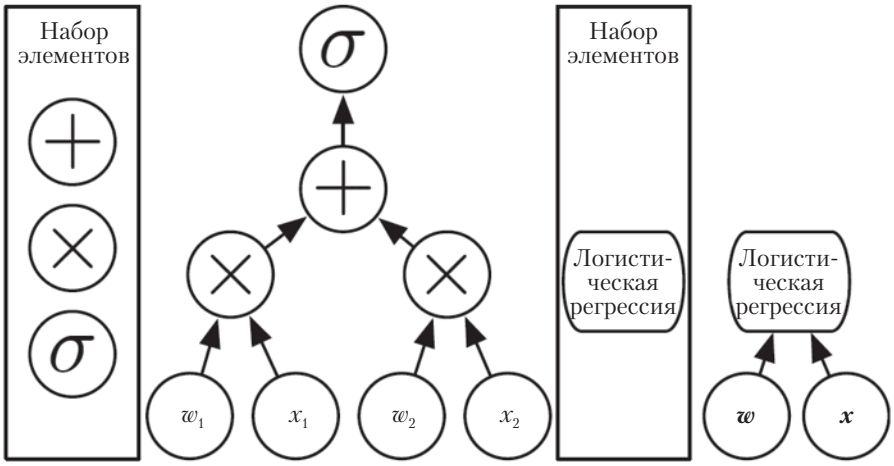

Рис. 1.3 ❖ Иллюстрация графов вычислений, переводящих вход в выход; в каждом узле выполняется некоторая операция. Глубиной считается длина самого длинного пути от входа к выходу, она зависит от определения допустимого шага вычисления. Оба графа описывают выход модели логистической регрессии $\sigma(\boldsymbol{w}^T\boldsymbol{x})$, где σ – логистическая сигмоида. Если в качестве элементов языка используются сложение, умножение и логистические сигмоиды, то глубина модели равна 3. Если же логистическая регрессия сама считается элементом языка, то глубина равна 1

Итак, глубокое обучение – тема этой книги – один из подходов к ИИ. Конкретно, это вид машинного обучения – методики, которая позволяет компьютерной системе совершенствоваться по мере накопления опыта и данных. Мы твердо убеждены, что машинное обучение – единственный жизнеспособный подход к построению систем ИИ, которые могут функционировать в сложных окружающих условиях. Глубокое обучение – это частный случай машинного обучения, позволяющий достичь большей эффективности и гибкости за счет представления мира в виде иерархии вложенных концепций, в которой каждая концепция определяется в терминах более простых концепций, а более абстрактные представления вычисляются в терминах менее абстрактных. На рис. 1.4 показано соотношение между разными отраслями ИИ, а на рис. 1.5 – высокоуровневое описание каждого подхода.

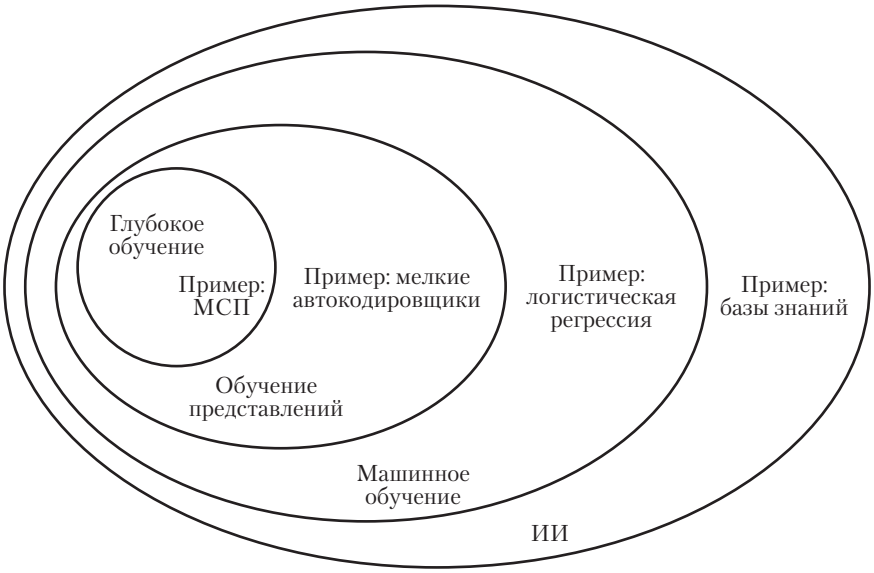

Рис. 1.4 ❖ На этой диаграмме Венна показано, что глубокое обучение – частный случай обучения представлений, которое, в свою очередь, является частным случаем машинного обучения, используемого во многих, но не во всех подходах к ИИ. На каждой части диаграммы Венна приведен пример технологии ИИ

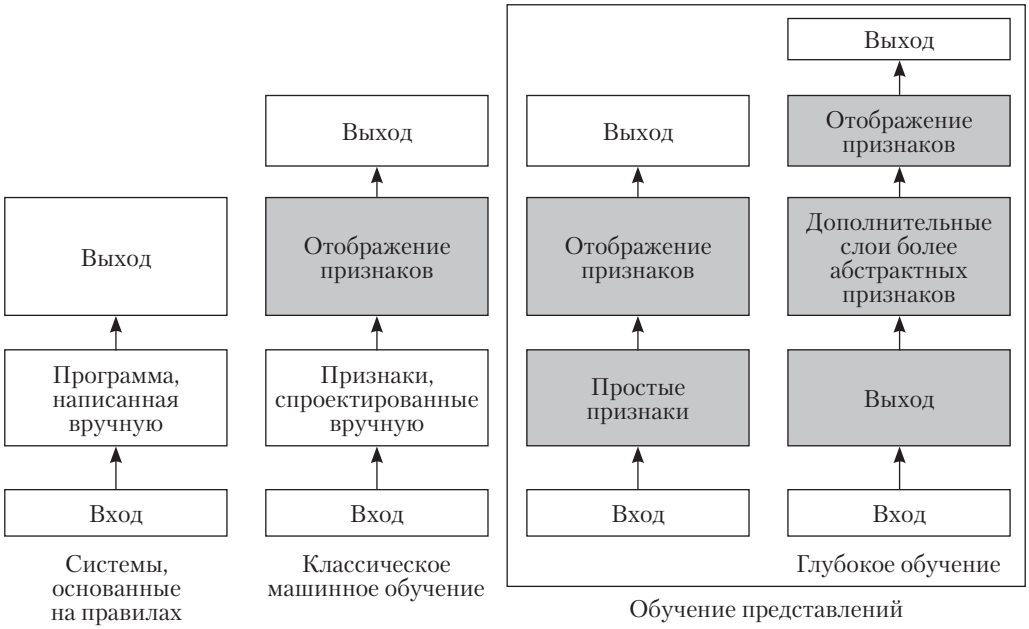

Рис. 1.5 ❖ Связь различных частей системы ИИ между собой в рамках разных подходов к ИИ. Серым цветом показаны компоненты, способные обучаться на данных

1.1. На кого ориентирована эта книга

Книга будет полезна разным читателям, но мы писали ее, имея в виду две целевые аудитории. Во-первых, это студенты университетов (как младших, так и старших курсов), изучающие машинное обучение, в том числе и те, кто начинает строить карьеру в области машинного обучения и искусственного интеллекта. Во-вторых, это разработчики ПО, которые не изучали машинное обучение или статистику, но хотят быстро освоить эти дисциплины и приступить к использованию глубокого обучения в своем продукте или платформе. Глубокое обучение доказало полезность во многих направлениях ПО, в т. ч. компьютерном зрении, распознавании речи и аудиозаписей, обработке естественных языков, робототехнике, биоинформатике и химии, видеоиграх, поисковых системах, интернет-рекламе и финансах.

Книга разделена на три части, чтобы лучше удовлетворить потребности разных категорий читателей. В части I излагается базовый математический аппарат и дается введение в концепции машинного обучения. В части II описаны самые известные алгоритмы глубокого обучения, которые можно считать сформировавшимися технологиями. Часть III посвящена более спорным идеям, которые многие считают важными для будущих исследований в области глубокого обучения.

Читатель может без ущерба для понимания пропускать части, не отвечающие его интересам или подготовке. Например, знакомые с линейной алгеброй, теорией вероятностей и фундаментальными концепциями машинного обучения могут пропустить часть I, а желающим реализовать работающую систему нет необходимости читать дальше части II. Чтобы читателю было проще понять, какие главы ему интересны, на рис. 1.6 показано, как устроена книга в целом.

Мы предполагаем, что читатели имеют подготовку в области информатики: знакомы с программированием, понимают, что такое производительность вычислений и теория сложности, знают начала математического анализа и некоторые положения и термины теории графов.

1.2. Исторические тенденции в машинном обучении

Понять глубокое обучение проще всего в историческом контексте. Мы не станем детально описывать историю глубокого обучения, а выделим несколько ключевых тенденций:

- у глубокого обучения долгая и богатая история, но оно фигурировало под разными названиями, отражающими различные философские воззрения, в связи с чем его популярность то усиливалась, то ослабевала;
- полезность глубокого обучения возросла, когда увеличился объем доступных обучающих данных;
- размер моделей глубокого обучения увеличивался по мере того, как совершенствовалась компьютерная инфраструктура (программная и аппаратная);
- со временем с помощью глубокого обучения удавалось решать все более сложные задачи со все большей точностью.

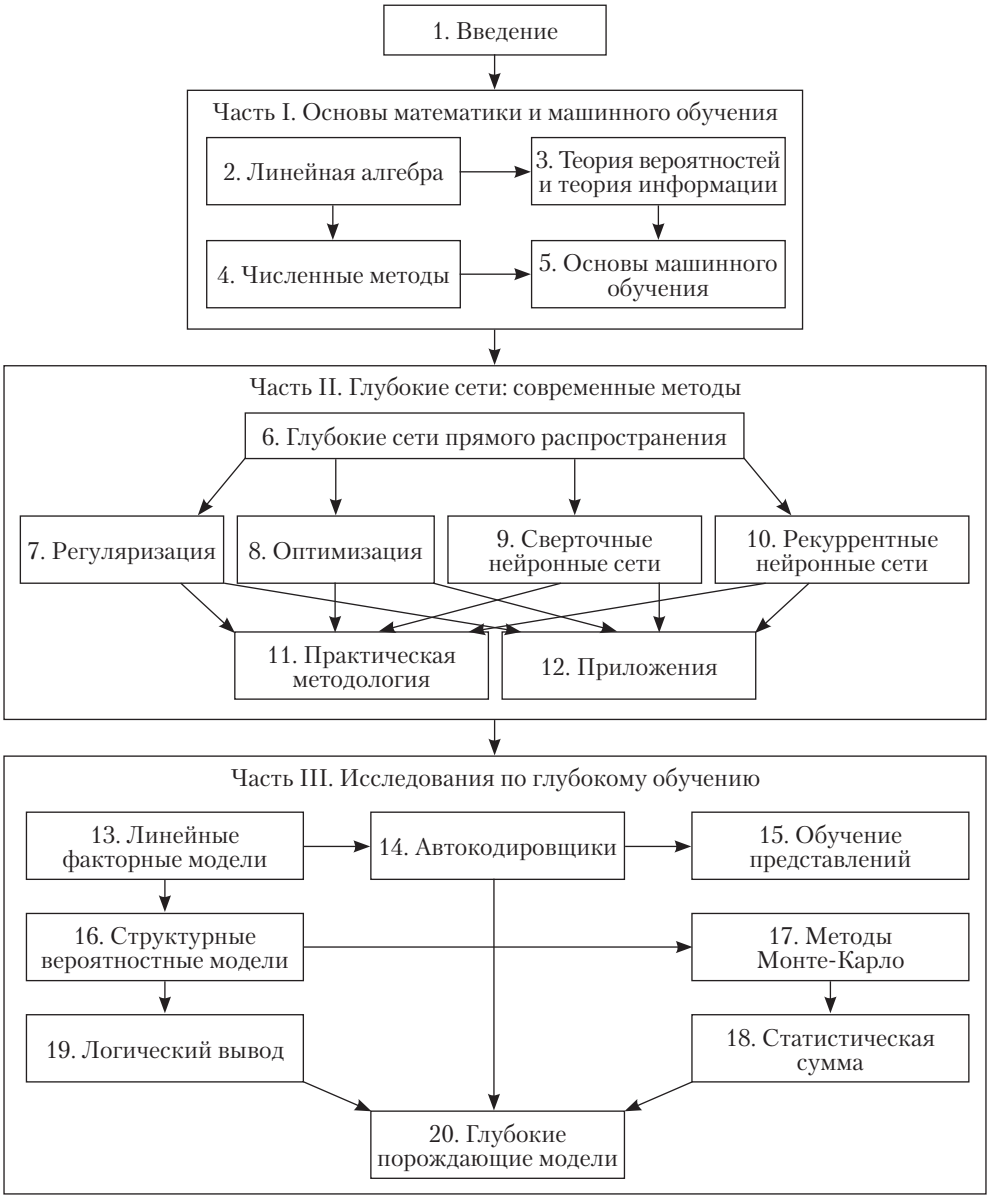

Рис. 1.6 ❖ Структура книги. Стрелки означают, что одна глава содержит материал, необходимый для понимания другой

1.2.1. Нейронные сети: разные названия и переменчивая фортуна

Мы полагаем, что многие читатели слышали о глубоком обучении как о сулящей чудеса новой технологии и удивлены, встретив слово «история» в применении к только зарождающейся дисциплине. Но в действительности глубокое обучение возникло еще в 1940-х годах. Оно кажется новым лишь потому, что в течение нескольких

лет, предшествующих нынешнему всплеску популярности, прозябало в тени, а также потому, что названия менялись, и только недавно эта дисциплина стала называться «глубоким обучением». Прежние названия отражали вес разных исследователей в научных кругах и разные точки зрения на предмет.

Изложение полной истории глубокого обучения выходит за рамки этого учебника. Но кое-какие базовые сведения помогут лучше понять его смысл. Если отвлечься от деталей, то было три волны разработок: в 1940–1960-х годах глубокое обучение было известно под названием **кибернетики**, в 1980–1990-х – как **коннекционизм**, а в современной инкарнации – под нынешним названием – оно возродилось в 2006 году. Количественная картина показана на рис. 1.7.

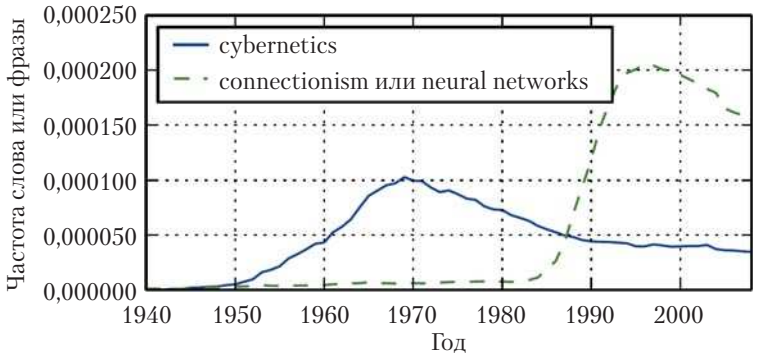

Рис. 1.7 ❖ Две из трех исторических волн исследований по искусственным нейронным сетям, оцененные по частоте фраз «cybernetics» и «connectionism или neural networks» согласно Google Books (третья волна началась недавно и еще не отражена). Первая волна, связанная с кибернетикой, приходится на 1940–1960-е годы, когда разрабатывались теории биологического обучения (McCulloch and Pitts, 1943; Hebb, 1949) и были реализованы первые модели, в частности перцептрон (Rosenblatt, 1958), позволявшие обучить один нейрон. Вторая волна периода 1980–1995 гг. связана с коннекционистским подходом, когда метод обратного распространения (Rumelhart et al., 1986a) был применен к обучению нейронной сети с одним или двумя скрытыми слоями. Третья волна, глубокое обучение, началась примерно в 2006 году (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007a) и только теперь – в 2016 году – описывается в виде книги. Книги, посвященные двум другим волнам, также вышли гораздо позже активизации исследований в соответствующей области

Некоторые из самых ранних алгоритмов обучения сегодня мы назвали бы компьютерными моделями биологического обучения, т. е. процессов, которые происходят или могли бы происходить в мозге. Поэтому одно из прежних названий глубокого обучения – искусственные нейронные сети (ИНС). Им соответствует взгляд на модели глубокого обучения как на инженерные системы, устроенные по образцу биологического мозга (человека или животного). Но хотя были попытки использовать нейронные сети, применяемые в машинном обучении, чтобы понять, как функционирует мозг (Hinton and Shallice, 1991), в общем случае при их проектировании не ставилась задача создать реалистическую модель биологической функции. Нейронный подход к глубокому обучению основан на двух главных идеях. Во-первых, мозг – доказатель-

ный пример того, что разумное поведение возможно. Поэтому концептуально прямой путь к построению искусственного интеллекта состоит в том, чтобы проанализировать с вычислительной точки зрения принципы работы мозга и воспроизвести его функциональность. Во-вторых, вообще было бы очень интересно понять, как работает мозг и какие принципы лежат в основе разума человека, поэтому модели машинного обучения, проливающие свет на эти фундаментальные научные вопросы, полезны вне зависимости от их применимости к конкретным инженерным задачам.

Современный термин «глубокое обучение» выходит за рамки нейробиологического взгляда на модели машинного обучения. В нем заложен более общий принцип обучения нескольких уровней композиции, применимый к системам машинного обучения, не обязательно устроенным по примеру нейронов.

Предтечами современного глубокого обучения были простые линейные модели на основе нейробиологических аналогий. Они принимали множество n входных значений x_1, ..., x_n и ассоциировали с ними выход y. В ходе обучения модель должна была найти веса w_1, ..., w_n и вычислить выход в виде $f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + ... + x_n w_n$. Эта первая волна нейронных сетей известна под названием кибернетики (рис. 1.7).

Нейрон Маккаллока–Питтса (McCulloch and Pitts, 1943) был ранней моделью функционирования мозга. Эта линейная модель могла распознавать две категории выходов, проверяя, является значение $f(\boldsymbol{x}, \boldsymbol{w})$ положительным или отрицательным. Конечно, чтобы модель соответствовала желаемому определению категорий, нужно было правильно подобрать веса. Веса задавал человек. В 1950-е годы был изобретен перцептрон (Rosenblatt, 1958, 1962) – первая модель, которая могла в процессе обучения находить веса, определяющие категории, имея примеры входных данных из каждой категории. Модель **адаптивного линейного элемента** (ADALINE), относящаяся примерно к тому же времени, просто возвращала само значение $f(\boldsymbol{x})$ для предсказания вещественного числа (Widrow and Hoff, 1960) и также могла обучаться предсказанию чисел на данных.

Эти простые алгоритмы обучения оказали заметное влияние на современный ландшафт машинного обучения. Алгоритм обучения, использованный для подбора весов в модели ADALINE, был частным случаем алгоритма **стохастического градиентного спуска**. Его немного модифицированные варианты и по сей день остаются основными алгоритмами моделей глубокого обучения.

Модели на основе функции $f(\boldsymbol{x}, \boldsymbol{w})$, используемые в перцептроне и ADALINE, называются **линейными моделями**. Они до сих пор относятся к числу наиболее распространенных моделей машинного обучения, хотя часто обучаются иначе, чем было принято раньше.

У линейных моделей много ограничений. Самое известное состоит в невозможности обучить функцию XOR, для которой $f([0, 1], \boldsymbol{w}) = 1$ и $f([1, 0], \boldsymbol{w}) = 1$, но $f([1, 1], \boldsymbol{w}) = 0$ и $f([0, 0], \boldsymbol{w}) = 0$. Критики, отмечавшие эти изъяны линейных моделей, вообще возражали против обучения, основанного на биологических аналогиях (Minsky and Papert, 1969). Это стало первым серьезным ударом по популярности нейронных сетей. В настоящее время нейробиология рассматривается как важный источник идей для исследований в области глубокого обучения, но уже не занимает доминирующих позиций.

Главная причина снижения роли нейробиологии в исследованиях по глубокому обучению – тот факт, что у нас попросту не хватает информации о мозге, чтобы использовать ее в качестве образца и руководства к действию. Чтобы по-настоящему

понять алгоритмы работы мозга, нужно было бы одновременно наблюдать за активностью тысяч (как минимум) взаимосвязанных нейронов. Не имея такой возможности, мы далеки от понимания даже самых простых и хорошо изученных областей мозга (Olshausen and Field, 2005).

Нейробиология дала надежду на то, что один алгоритм глубокого обучения сможет решить много разных задач. Нейробиологи выяснили, что хорьки могут «видеть» с помощью области мозга, отвечающей за обработку слуховой информации, если перенаправить нервы из глаз в слуховую кору (Von Melchner et al., 2000). Это наводит на мысль, что значительная часть мозга млекопитающих, возможно, использует единый алгоритм для решения большинства задач. До появления этой гипотезы исследования по машинному обучению были более фрагментированы: обработкой естественных языков, компьютерным зрением, планированием движения и распознаванием речи занимались разные группы ученых. Эти сообщества и по сей день разделены, но ученые, работающие в области глубокого обучения, зачастую занимаются многими или даже всеми этими предметами.

Мы можем почерпнуть из нейробиологии кое-какие соображения. Основная идея, навеянная осмыслением работы мозга, – наличие большого числа вычислительных блоков, которые обретают разум только в результате взаимодействий. Неокогнитрон (Fukushima, 1980) предложил архитектуру модели, эффективную для обработки изображений. Идея сложилась под влиянием структуры зрительной системы млекопитающих и впоследствии легла в основу современной сверточной сети (LeCun et al., 1998b), с которой мы познакомимся в разделе 9.10. Большинство современных нейронных сетей основано на модели нейрона, которая называется **блоком линейной ректификации**. Оригинальная модель когнитрона (Fukushima, 1975) была более сложной, основанной на наших знаниях о работе мозга. В современной упрощенной модели объединены различные точки зрения; в работах Nair and Hinton (2010) и Glorot et al. (2011a) отмечено влияние нейробиологии, а в работе Jarrett et al. – скорее, инженерных дисциплин. Каким бы важным источником идей ни была нейробиология, считать, что от нее нельзя отклониться ни на шаг, вовсе необязательно. Мы знаем, что настоящие нейроны вычисляют совсем не те функции, что блоки линейной ректификации, но большее приближение к реальности пока не привело к повышению качества машинного обучения. Кроме того, хотя нейробиология легла в основу нескольких успешных архитектур нейронных сетей, мы до сих пор знаем о биологическом обучении недостаточно, чтобы полнее использовать нейробиологию для построения *алгоритмов обучения* этих архитектур.

В социальных сетях часто подчеркивают сходство между глубоким обучением и мозгом. Да, действительно, исследователи, занимающиеся глубоким обучением, чаще говорят о влиянии аналогий с мозгом на свою работу, чем те, кто занимается другими методами машинного обучения, например ядерными методами или байесовской статистикой. Но не следует думать, что цель глубокого обучения – попытка имитировать мозг. Современное глубокое обучение черпает идеи из разных дисциплин, особенно из таких отраслей прикладной математики, как линейная алгебра, теория вероятностей, теория информации и численная оптимизация. Некоторые ученые считают нейробиологию важным источником идей, другие не упоминают ее вовсе.

Отметим, что попытки понять, как работает мозг на алгоритмическом уровне, не прекращаются. Эта область исследований, известная под названием «вычислительная нейробиология», не совпадает с глубоким обучением. Нередко ученые

переходят из одной области в другую. Предмет глубокого обучения – построение компьютерных систем, способных успешно решать задачи, требующие интеллекта, а предмет вычислительной нейробиологии – построение более точных моделей работы мозга.

В 1980-е годы поднялась вторая волна исследований по нейронным сетям, вызванная главным образом движением под названием **коннекционизм**, или **параллельная распределенная обработка** (Rumelhart et al., 1986c; McClelland et al., 1995). Коннекционизм возник в контексте когнитивистики – междисциплинарного подхода к пониманию процесса познания, объединяющего несколько разных уровней анализа. В начале 1980-х годов большинство когнитивистов изучало модели принятия решений путем манипулирования символами (symbolic reasoning). Несмотря на популярность символических моделей, трудно было объяснить, как мозг мог бы реализовать их с помощью нейронов. Коннекционисты начали изучать модели познания, которые допускали реализацию на основе нейронов (Touretzky and Minton, 1985), возродив многие идеи психолога Дональда Хебба, высказанные в 1940-х годах (Hebb, 1949).

Центральная идея коннекционизма состоит в том, что при наличии большого количества вычислительных блоков, объединенных в сеть, удастся достичь разумного поведения. Эта идея относится в равной мере к нейронам в биологических нервных системах и к скрытым блокам в компьютерных моделях.

Движение коннекционизма породило несколько ключевых концепций, которые и по сей день играют важнейшую роль в глубоком обучении.

Одна из них – **распределенное представление** (Hinton et al., 1986). Идея в том, что каждый вход системы следует представлять многими признаками, а каждый признак должен участвовать в представлении многих возможных входов. Пусть, например, имеется зрительная система, способная распознавать легковые автомобили, грузовики и птиц, причем объекты могут быть красного, зеленого или синего цвета. Один из способов представления таких входов – завести отдельный нейрон или скрытый блок для активации каждой из девяти возможных комбинаций: красный грузовик, красная легковушка, красная птица, зеленый грузовик и т. д. Тогда потребуется девять нейронов, и каждый нейрон необходимо независимо обучить концепциям цвета и типа объекта. Улучшить ситуацию можно, воспользовавшись распределенным представлением, в котором три нейрона описывают цвет, а еще три – тип объекта. Тогда понадобится всего шесть нейронов, и нейрон, отвечающий за красное, можно обучить на изображениях легковушек, грузовиков и птиц, а не только на изображениях объектов одного типа. Концепция распределенного представления является центральной в этой книге и подробно рассматривается в главе 15.

Еще одним крупным достижением коннекционистов стали успешное использование обратного распространения для обучения глубоких нейронных сетей с внутренними представлениями и популяризация алгоритма обратного распространения (Rumelhart et al., 1986a; LeCun, 1987). Его популярность то возрастала, то убывала, но на данный момент это преобладающий подход к обучению глубоких моделей.

1990-е годы стали временем важных достижений в моделировании последовательностей с помощью нейронных сетей. В работах Hochreiter (1991) и Bengio et al. (1994) сформулирован ряд фундаментальных математических трудностей моделирования длинных последовательностей (см. раздел 10.7). В работе Hochreiter and Schmidhuber (1997) введено понятие сетей с долгой краткосрочной памятью (long short-term me-

mory – LSTM) для разрешения некоторых из описанных трудностей. Сегодня LSTM-сети широко используются во многих задачах моделирования последовательностей, в т. ч. для обработки естественных языков в Google.

Вторая волна работ по нейронным сетям продолжалась до середины 1990-х годов. Но компании, специализирующиеся на нейронных сетях и других технологиях ИИ, стали давать чрезмерно амбициозные обещания в попытках привлечь инвестиции. Когда ИИ не оправдал этих неразумных надежд, инвесторы испытали разочарование. В то же время имел место заметный прогресс в других областях машинного обучения. Ядерные методы (Boser et al., 1992; Cortes and Vapnik, 1995; Schölkopf et al., 1999) и графические модели (Jordan, 1998) позволили достичь хороших результатов при решении многих важных задач. В совокупности эти два фактора привели к спаду интереса к нейронным сетям, который продолжался до 2007 года.

В это время нейронные сети по-прежнему показывали впечатляющее качество на некоторых задачах (LeCun et al., 1998b; Bengio et al., 2001). Канадский институт перспективных исследований (Canadian Institute for Advanced Research – CIFAR) помог нейронным сетям остаться на плаву, профинансировав исследовательскую программу нейронных вычислений и адаптивного восприятия (Neural Computation and Adaptive Perception – NCAP). В рамках этой программы объединились группы Джеффри Хинтона из Торонтского университета, Иошуа Бенджио из Монреальского университета и Янна Лекуна из Нью-Йоркского университета. Мультидисциплинарная исследовательская программа CIFAR NCAP включала также нейробиологов и специалистов по человеческому и компьютерному зрению.

Тогда сложилось общее мнение, что обучить глубокие сети очень трудно. Теперь мы знаем, что алгоритмы, существовавшие начиная с 1980-х годов, работают отлично, но это не было очевидно до 2006 года. Причина, наверное, в том, что с вычислительной точки зрения эти алгоритмы очень накладны, поэтому было невозможно экспериментировать с ними на имевшемся тогда оборудовании.

Третья волна работ по нейронным сетям началась с прорыва в 2006 году. Джеффри Хинтон показал, что так называемые **глубокие сети доверия** можно эффективно обучать с помощью стратегии жадного послойного предобучения (Hinton et al., 2006), которую мы подробно опишем в разделе 15.1. Другие аффилированные с CIFAR исследовательские группы быстро показали, что ту же стратегию можно использовать для обучения многих других видов глубоких сетей (Bengio et al., 2007; Ranzato et al., 2007a), и систематически улучшали степень обобщения на тестовых примерах. Благодаря этой волне исследований в обиход вошел термин «глубокое обучение», подчеркивающий, что теперь можно обучать более глубокие нейронные сети, чем раньше, и привлекающий внимание к теоретической важности глубины (Bengio and LeCun, 2007; Delalleau and Bengio, 2011; Pascanu et al., 2014a; Montufar et al., 2014). В то время глубокие нейронные сети превзошли конкурирующие системы ИИ – как основанные на других технологиях машинного обучения, так и спроектированные вручную. Третья волна популярности нейронных сетей продолжается и во время написания этой книги, хотя фокус исследований значительно сместился. Поначалу в центре внимания находились методы обучения без учителя и способность глубоких моделей, обученных на небольших наборах данных, к обобщению. А теперь больший интерес вызывают гораздо более старые алгоритмы обучения с учителем и возможность задействовать большие размеченные наборы данных при обучении глубоких моделей.

1.2.2. Увеличение размера набора данных

Может возникнуть вопрос, почему лишь недавно была осознана роль глубокого обучения как ключевой технологии, хотя первые эксперименты с искусственными нейронными сетями были проведены еще в 1950-х годах. Глубинное обучение успешно применяется в коммерческих приложениях, начиная с 1990-х, но часто его рассматривали не как технологию, а как искусство – как нечто такое, что подвластно только экспертам. Так было до недавнего времени. Действительно, чтобы добиться хорошего качества от алгоритма глубокого обучения, нужен некоторый навык. Но, к счастью, потребность в таком навыке снижается по мере увеличения объема обучающих данных. Алгоритмы обучения, по качеству приближающиеся к возможностям человека при решении сложных задач, остались почти такими же, как алгоритмы, с трудом справлявшиеся с игрушечными задачами в 1980-х, но модели, которые мы с их помощью обучаем, претерпели изменения, позволившие упростить обучение очень глубоких архитектур. Самое важное новшество заключается в том, что сегодня мы можем предоставить этим алгоритмам необходимые для успеха ресурсы. На рис. 1.8 показано изменение эталонных наборов данных со временем. Эта тенденция обусловлена возрастающей цифровизацией общества. Чем активнее применяются компьютеры, тем больше записей о том, что мы делаем. А поскольку компьютеры объединяются в сети, становится проще централизованно хранить эти записи и построить из них набор данных, подходящий для машинного обучения. С наступлением эры «больших данных» машинное обучение значительно упростилось, поскольку ключевая проблема статистического оценивания – высокое качество обобщения на новые данные после обучения на небольшом количестве примеров – теперь далеко не так актуальна. В 2016 году действует грубое эвристическое правило: алгоритм глубокого обучения с учителем достигает приемлемого качества при наличии примерно 5000 помеченных примеров на категорию и оказывается сопоставим или даже превосходит человека, если обучается на наборе данных, содержащем не менее 10 миллионов помеченных примеров. Как добиться успеха при работе с наборами данных меньшего размера – важная область исследований, и акцент в ней делается на том, чтобы воспользоваться преимуществами большого количества непомеченных примеров, применяя обучение без учителя или с частичным привлечением учителя.

1.2.3. Увеличение размера моделей

Еще одна причина нынешнего роста популярности нейронных сетей после сравнительно скромных успехов в 1980-е годы – наличие вычислительных ресурсов, достаточных для работы с гораздо более крупными моделями. Один из главных выводов коннекционизма состоит в том, что животные проявляют интеллект, когда много нейронов работает совместно. Один нейрон или небольшой их набор не принесет особой пользы.

Плотность соединений между биологическими нейронами не слишком велика. Как видно по рис. 1.10, в наших моделях машинного обучения число соединений в расчете на один нейрон примерно на порядок выше, чем даже в мозгу млекопитающих, и такое положение существует уже несколько десятков лет.

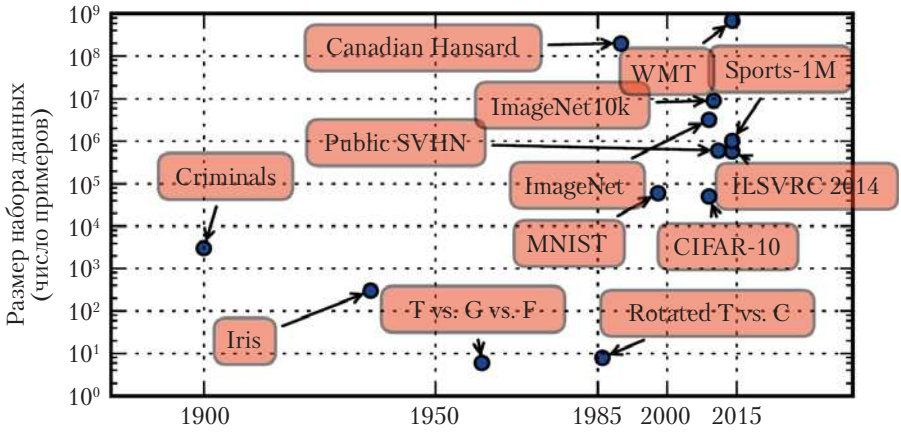

Рис. 1.8 ❖ Увеличение размера набора данных со временем. В начале 1900-х годов статистики изучали наборы данных, содержащие от сотен до тысяч вручную подготовленных измерений (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). В период между 1950-ми и 1980-ми пионеры биокомпьютерного зрения зачастую работали с небольшими синтетическими наборами данных, например растровыми изображениями букв низкого разрешения, специально спроектированными так, чтобы снизить стоимость вычислений и продемонстрировать, что нейронные сети можно обучить функциям специального вида (Widrow and Hoff, 1960; Rumelhart et al., 1986b). В 1980-е и 1990-е машинное обучение стало в большей степени статистическим, а наборы данных уже насчитывали десятки тысяч примеров, как, например, набор MNIST (рис. 1.9) отсканированных рукописных цифр (LeCun et al., 1998b). В первом десятилетии XXI века продолжали создавать более изощренные наборы данных того же размера, например CIFAR-10 (Krizhevsky and Hinton, 2009). В конце этого периода и в первой половине 2010-х появление гораздо больших наборов данных, содержащих от сотен тысяч до десятков миллионов примеров, полностью изменило представление о возможностях глубокого обучения. К таким наборам относится общедоступный набор номеров домов (Street View House Numbers) (Netzer et al., 2011), различные варианты набора ImageNet (Deng et al., 2009, 2010a; Russakovsky et al., 2014a) и набор Sports-1M (Karpathy et al., 2014). В верхней части диаграммы мы видим, что наборы переведенных предложений, например набор, построенный IBM по официальным отчетам о заседаниях канадского парламента (Brown et al., 1990), и набор WMT 2014 переводов с английского на французский (Schwenk, 2014), по размеру намного превосходят большинство остальных наборов

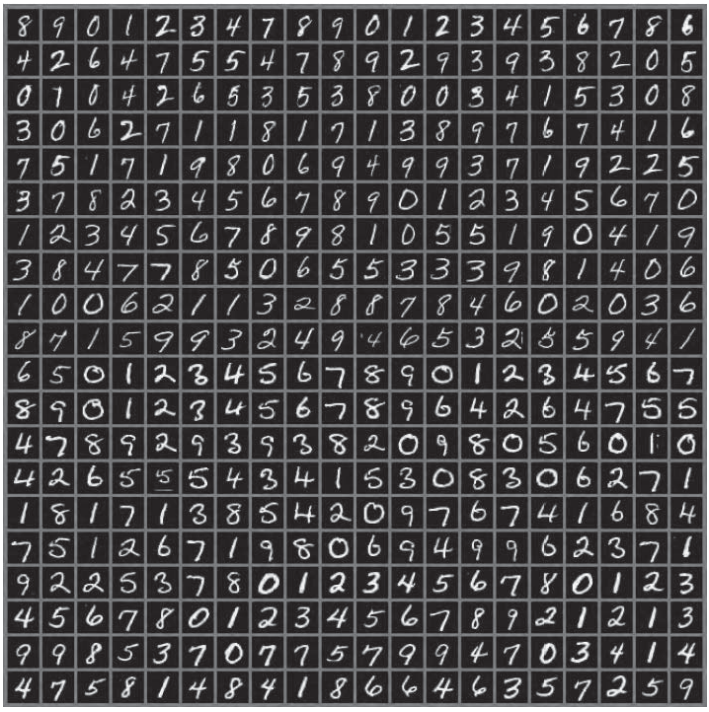

Рис. 1.9 ❖ Несколько примеров из набора данных MNIST. Акроним «NIST» означает «National Institute of Standards and Technology» (Национальный институт стандартов и технологий) – учреждение, первоначально собравшее эти данные. А буква «M» означает «модифицированный», поскольку данные были подвергнуты предварительной обработке, чтобы упростить применение алгоритмов машинного обучения. Набор данных MNIST содержит отсканированные изображения рукописных цифр и ассоциированные с ними метки, описывающие, какая цифра от 0 до 9 изображена. Это одна из самых простых задач классификации, поэтому набор широко используется для тестирования в исследованиях по глубокому обучению. Он сохраняет популярность, хотя для современных методов задача не представляет труда. Джеффри Хинтон назвал его «дрозофилой машинного обучения», имея в виду, что он позволяет специалистам по машинному обучению исследовать свои алгоритмы в контролируемых лабораторных условиях, как биологи изучают фруктовых мушек

Если говорить об общем числе нейронов, то до недавнего времени нейронные сети были на удивление малы (рис. 1.11). После добавления скрытых блоков размер искусственных нейронных сетей удваивался в среднем каждые 2,4 года, чему способствовало появление более быстрых компьютеров с большим объемом памяти, а также наличие крупных наборов данных. Чем больше сеть, тем выше ее точность на более сложных задачах. Эта тенденция прослеживается на протяжении десятилетий. Если темпы масштабирования не ускорятся в результате внедрения новых технологий, то искусственная нейронная сеть сравняется с человеческим мозгом по числу нейронов

не раньше 2050-х годов. Биологические нейроны могут представлять более сложные функции, чем современные искусственные, поэтому биологические нейронные сети могут оказаться даже больше, чем показано на диаграмме.

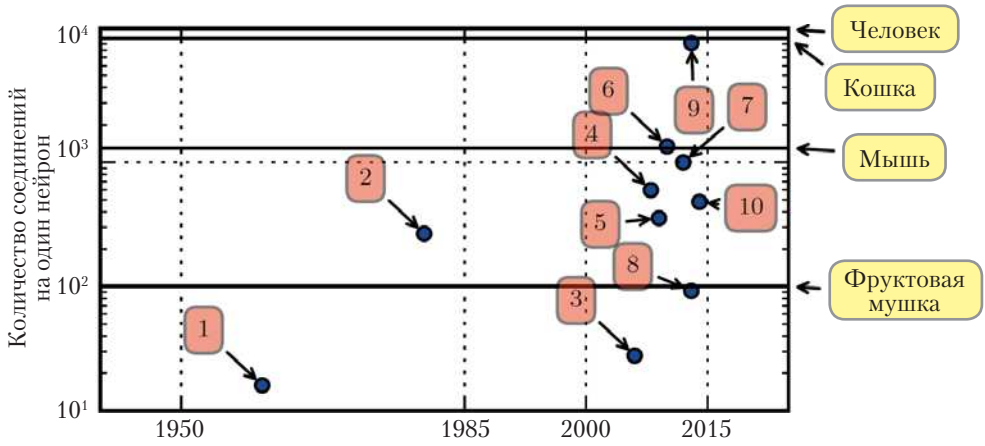

Рис. 1.10 ❖ Рост количества соединений в расчете на один нейрон со временем. Первоначально число соединений между нейронами в искусственной нейронной сети было ограничено возможностями оборудования. Сегодня оно в основном является параметром проектирования. В некоторых нейронных сетях число соединений почти такое же, как у кошки, а сети с числом соединений, как у мелких млекопитающих типа мыши, встречаются сплошь и рядом. Даже в мозге человека число соединений на нейрон нельзя назвать заоблачным. Размеры биологических нейронных сетей взяты из Википедии (2015)

1. Адаптивный линейный элемент (Widrow and Hoff, 1960)
2. Неокогнитрон (Fukushima, 1980)
3. GPU-ускоренная сверточная сеть (Chellapilla et al., 2006)
4. Глубокая машина Больцмана (Salakhutdinov and Hinton, 2009a)
5. Сверточная сеть без учителя (Jarrett et al., 2009)
6. GPU-ускоренный многослойный перцептрон (Ciresan et al., 2010)
7. Распределенный автокодировщик (Le et al., 2012)
8. Сверточная сеть с несколькими GPU (Krizhevsky et al., 2012)
9. Сверточная сеть без учителя на компьютерах типа COTS HPC (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

В ретроспективе не кажется удивительным, что нейронные сети с числом нейронов меньше, чем у пиявки, не могли справиться с решением сложных задач искусственного интеллекта. Даже нынешние сети, которые мы считаем очень большими с вычислительной точки зрения, меньше нервной системы сравнительно примитивных позвоночных животных, например лягушек.

Увеличение размера моделей вследствие доступности более быстрых процессоров, GPU общего назначения (см. раздел 12.1.2), более быстрых сетей и более совершенной инфраструктуры распределенных вычислений – одна из самых важных тенденций в истории глубокого обучения. Ожидается, что она продолжится и в будущем.

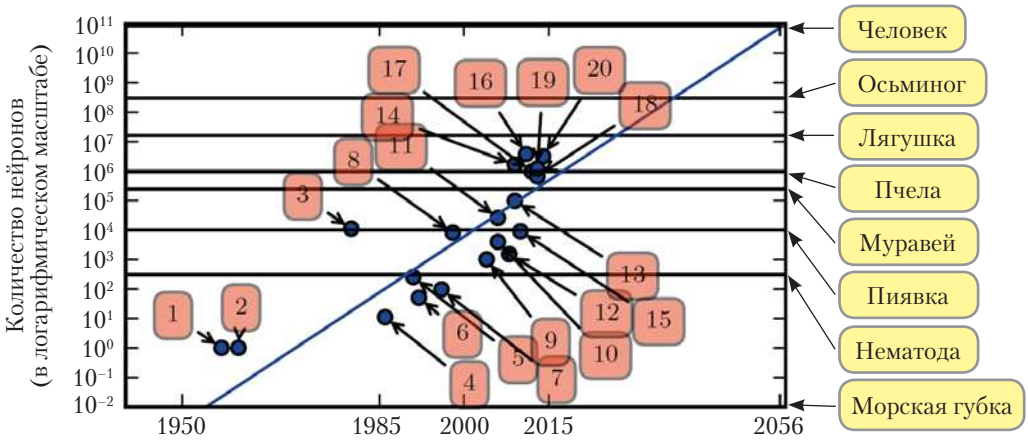

Рис. 1.11 ❖ Рост размера нейронной сети со временем. После добавления скрытых блоков размер нейронных сетей удваивался примерно каждые 2,4 года. Размеры биологических нейронных сетей взяты из Википедии (2015)

1. Перцептрон (Rosenblatt, 1958, 1962)
2. Адаптивный линейный элемент (Widrow and Hoff, 1960)
3. Неокогнитрон (Fukushima, 1980)
4. Ранняя сеть с обратным распространением (Rumelhart et al., 1986b)
5. Рекуррентная нейронная сеть для распознавания речи (Robinson and Fallside, 1991)
6. Многослойный перцептрон для распознавания речи (Bengio et al., 1991)
7. Сигмоидальная сеть доверия со средним полем (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Нейронная эхо-сеть (Jaeger and Haas, 2004)
10. Глубокая сеть доверия (Hinton et al., 2006)
11. GPU-ускоренная сверточная сеть (Chellapilla et al., 2006)
12. Глубокая машина Больцмана (Salakhutdinov and Hinton, 2009a)
13. GPU-ускоренная сеть глубокого доверия (Raina et al., 2009)
14. Сверточная сеть без учителя (Jarrett et al., 2009)
15. GPU-ускоренный многослойный перцептрон (Ciresan et al., 2010)
16. Сеть OMP-1 (Coates and Ng, 2011)
17. Распределенный автокодировщик (Le et al., 2012)
18. Сверточная сеть с несколькими GPU (Krizhevsky et al., 2012)
19. Сверточная сеть без учителя на компьютерах типа COTS HPC (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)

1.2.4. Повышение точности и сложности и расширение круга задач

Начиная с 1980-х годов точность распознавания и прогнозирования глубоких моделей постоянно росла. И вместе с тем все разнообразнее становились задачи, которые удавалось решать с их помощью.

Самые первые глубокие модели использовались для распознавания отдельных объектов в кадрированных изображениях совсем небольшого размера (Rumelhart et al., 1986a). С тех пор размер изображений, которые можно было обработать с помощью нейронной сети, постепенно увеличивался. Современные сети распознавания объектов обрабатывают фотографии с высоким разрешением и не требуют кадрирования фотографии по месту расположения объекта (Krizhevsky et al., 2012). Кроме того, ранние сети умели распознавать только два вида объектов (а в некоторых случаях при-

сутствие или отсутствие объектов одного вида), тогда как типичная современная сеть распознает не менее 1000 категорий объектов. Самый крупный конкурс по распознаванию объектов – ImageNet Large Scale Visual Recognition Challenge (ILSVRC) – проводится каждый год. Переломным моментом, ознаменовавшим стремительный взлет глубокого обучения, стала победа с большим отрывом сверточной сети, которая участвовала впервые и сразу уменьшила частоту непопадания в первые пять (top-5 error rate) с 26,1 до 15,3% (Krizhevsky et al., 2012). Смысл этого показателя следующий: сверточная сеть порождала для каждого изображения ранжированный список возможных категорий, и правильная категория отсутствовала среди первых пяти элементов этого списка только в 15,3% тестовых примеров. С тех пор подобные конкурсы неизменно выигрывали сверточные сети, и на данный момент прогресс глубокого обучения позволил довести частоту непопадания в первые пять до 3,6% (рис. 1.12).

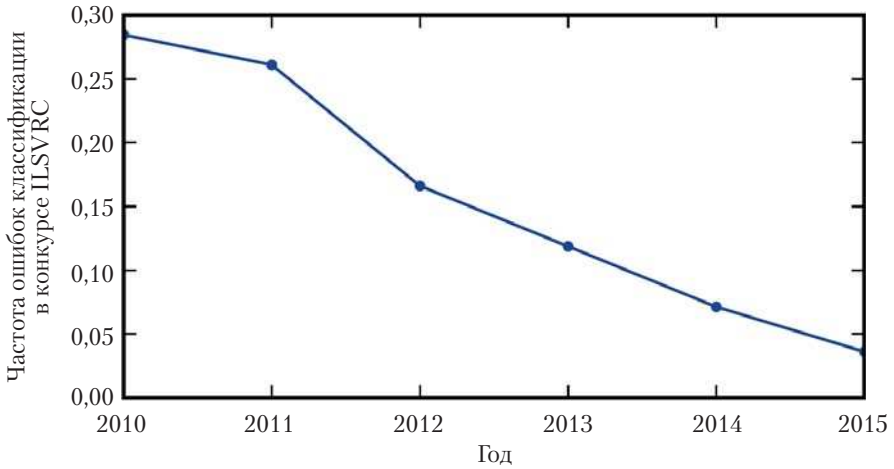

Рис. 1.12 ❖ Уменьшение частоты ошибок со временем. С тех пор как глубокие сети достигли масштаба, необходимого для участия в конкурсе ImageNet Large Scale Visual Recognition Challenge, они неизменно выигрывают его, с каждым разом демонстрируя все меньшую и меньшую частоту ошибок. Данные взяты из работ Russakovsky et al. (2014b) и He et al. (2015)

Глубокое обучение также оказало огромное влияние на распознавание речи. После прогресса, достигнутого на протяжении 1990-х годов, в качестве распознавания речи наступил застой. Применение глубокого обучения (Dahl et al., 2010; Deng et al., 2010b; Seide et al., 2011; Hinton et al., 2012a) привело к резкому уменьшению частоты ошибок, иногда аж наполовину. Мы вернемся к этой теме в разделе 12.3.

Глубокие сети добились также впечатляющих успехов в обнаружении пешеходов и сегментации изображений (Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013), а в задаче классификации дорожных знаков показали себя лучше человека (Ciresan et al., 2012).

Одновременно с повышением размера и точности глубоких сетей росла и сложность решаемых с их помощью задач. В работе Goodfellow et al. (2014d) показано, что нейронные сети можно научить распознаванию целых последовательностей символов в изображении, а не только идентификации одиночного объекта. Ранее считалось, что для такого обучения необходимо помечать отдельные элементы последовательности (Gülçehre

and Bengio, 2013). Рекуррентные нейронные сети, в частности вышеупомянутая модель LSTM, теперь применяются для моделирования связей *последовательностей* с другими *последовательностями*, а не только с фиксированными входами. Такое обучение типа «последовательность в последовательность», похоже, привело к революции в другом приложении: машинном переводе (Sutskever et al., 2014; Bahdanau et al., 2015).

Эта тенденция к увеличению сложности была доведена до логического завершения с вводом в рассмотрение нейронных машин Тьюринга (Graves et al., 2014a), которые обучаются читать из ячеек памяти и писать произвольные данные в ячейки памяти. Такие нейронные сети могут обучаться простым программам на примерах желаемого поведения. Например, они могут научиться сортировать списки чисел, если им предъявить примеры отсортированных и неотсортированных последовательностей. Эта технология самопрограммирования пока находится в зачаточной стадии, но в будущем теоретически может быть применена почти к любой задаче.

Еще одним венцом глубокого обучения является обобщение на **обучение с подкреплением**. В этом контексте автономный агент должен научиться выполнять некоторое задание методом проб и ошибок, без какой-либо подсказки со стороны человека. Компания DeepMind продемонстрировала систему обучения с подкреплением, основанную на технологиях глубокого обучения, которая способна научиться играть в видеоигры Atari, достигая во многих случаях уровня, сравнимого с человеческим (Mnih et al., 2015). Глубокое обучение позволило также значительно усовершенствовать качество обучения с подкреплением в робототехнике (Finn et al., 2015).

Многие приложения глубокого обучения приносят солидную прибыль. Эти технологии используются в таких крупных компаниях, как Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA и NEC.

Прогресс в глубоком обучении сильно зависит от достижений в области программной инфраструктуры. Библиотеки Theano (Bergstra et al., 2010; Bastien et al., 2012), PyLearn2 (Goodfellow et al., 2013c), Torch (Collobert et al., 2011b), DistBelief (Dean et al., 2012), Caffe (Jia, 2013), MXNet (Chen et al., 2015) и TensorFlow (Abadi et al., 2015) применяются в важных исследовательских проектах и в коммерческих продуктах.

Глубокое обучение внесло вклад и в другие науки. Современные сверточные сети для распознавания объектов дают модель визуальной обработки, изучаемую в нейробиологии (DiCarlo, 2013). Кроме того, глубокое обучение предоставляет полезные инструменты для обработки больших массивов данных и прогнозирования в различных научных дисциплинах. Оно было успешно применено к прогнозированию взаимодействия молекул в интересах фармацевтических компаний, занимающихся поиском новых лекарств (Dahl et al., 2014), к поиску субатомных частиц (Baldi et al., 2014) и к автоматическому распознаванию сделанных микроскопом изображений для построения трехмерной карты человеческого мозга (Knowles-Barley et al., 2014). В будущем мы ожидаем проникновения глубокого обучения во все новые и новые отрасли науки.

Резюмируя, можно сказать, что глубокое обучение – это развивавшийся в течение нескольких десятилетий подход к машинному обучению, основанный главным образом на наших знаниях о человеческом мозге, на статистике и прикладной математике. В последние годы глубокое обучение переживает стремительный рост популярности и полезности благодаря в первую очередь появлению более мощных компьютеров, больших наборов данных и методов обучения более глубоких сетей. Будущее сулит нам новые проблемы и возможности дальнейшего совершенствования глубокого обучения с выходом на новые рубежи.

Часть I

ОСНОВЫ ПРИКЛАДНОЙ МАТЕМАТИКИ И МАШИННОГО ОБУЧЕНИЯ

В этой части книги вводятся основные математические идеи, необходимые для понимания глубокого обучения. Мы начнем с общего математического аппарата: определения функций многих переменных, нахождения их минимумов и максимумов и численной оценки степени доверия.

Затем мы сформулируем фундаментальные цели машинного обучения и опишем, как их достичь с помощью задания модели, представляющей некоторые гипотезы, проектирования функции стоимости, которая измеряет степень соответствия гипотез реальности, и использования алгоритма обучения для минимизации функции стоимости.

Этот элементарный аппарат является основной для самых разнообразных алгоритмов машинного обучения, не обязательно глубокого. В последующих частях книги мы с его помощью разработаем алгоритмы глубокого обучения.

Глава 2

Линейная алгебра

Линейная алгебра – это раздел математики, который широко используется в науке и технике. Но поскольку линейная алгебра имеет дело с непрерывными, а не дискретными величинами, специалисты по информатике редко с ней сталкиваются. Однако для понимания многих алгоритмов машинного обучения, особенно глубокого, уверенное владение аппаратом линейной алгебры необходимо. Поэтому мы предпошлем введению в глубокое обучение концентрированное изложение основ линейной алгебры.

Если вы уже знакомы с линейной алгеброй, то можете пропустить эту главу. Если вам раньше доводилось встречаться с этим аппаратом, но нужен подробный справочник, чтобы освежить в памяти формулы, то рекомендуем книгу «The Matrix Cookbook» (Petersen and Pedersen, 2006). Если вы совсем ничего не знаете о линейной алгебре, то из этой главы вы почерпнете достаточно сведений для чтения книги, но мы настоятельно советуем обратиться к еще какому-нибудь источнику, посвященному исключительно линейной алгебре, например книге Шилова, Shilov (1977). В этой главе не рассматриваются многие вопросы линейной алгебры, не существенные для понимания глубокого обучения.

2.1. Скаляры, векторы, матрицы и тензоры

В линейной алгебре изучаются математические объекты нескольких видов.

- **Скаляры.** Скаляр – это просто число, в отличие от большинства других изучаемых в линейной алгебре объектов, представляющих собой массивы чисел. Мы будем обозначать скаляры курсивным шрифтом и, как правило, строчными буквами. Вводя в рассмотрение скаляр, мы будем указывать, что он означает, например: «Пусть $s \in \mathbb{R}$ обозначает угловой коэффициент прямой» (вещественное число) или «Обозначим $n \in \mathbb{N}$ число блоков» (целое натуральное число).
- **Векторы.** Вектор – это массив чисел. Элементы вектора расположены в определенном порядке. Отдельный элемент определяется своим индексом. Обычно векторы обозначаются строчными буквами курсивным полужирным шрифтом, например $\boldsymbol{x}$. Для обозначения элемента вектора указывается имя вектора курсивным шрифтом с подстрочным индексом. Первый элемент вектора $\boldsymbol{x}$ обозначается x_1, второй – x_2 и т. д. Необходимо также указать тип хранящихся в векторе чисел. Если каждый элемент вектора принадлежит $\mathbb{R}$, а всего элементов n, то вектор принадлежит декартову произведению n экземпляров $\mathbb{R}$, оно обозначается $\mathbb{R}^n$. Если требуется явно перечислить элементы вектора, то они записываются в столбик в квадратных скобках:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \tag{2.1}$$

Можно считать, что векторам соответствуют точки в пространстве, а каждый элемент – координата вдоль одной из осей.

Иногда необходимо обозначить подмножество элементов вектора. В таком случае мы будем указывать в качестве подстрочного индекса множество индексов интересующих элементов. Например, чтобы адресовать элементы x_1, x_3 и x_6, мы определим множество $S = \{1, 3, 6\}$ и будем писать $\boldsymbol{x}_S$. Для обозначения дополнения ко множеству будем использовать знак –. Например, $\boldsymbol{x}_{-1}$ обозначает вектор, содержащий все элементы $\boldsymbol{x}$, кроме x_1, а $\boldsymbol{x}_{-S}$ – вектор, содержащий все элементы $\boldsymbol{x}$, кроме x_1, x_3 и x_6.

- **Матрицы.** Матрица – это двумерный массив чисел, в котором каждый элемент идентифицируется двумя индексами, а не одним. Обычно матрицы обозначаются прописными буквами полужирным курсивным шрифтом, например $\boldsymbol{A}$. Если матрица вещественных чисел $\boldsymbol{A}$ имеет высоту m и ширину n, то говорят, что $\boldsymbol{A} \in \mathbb{R}^{m \times n}$. Элементы матрицы обычно обозначаются светлым курсивным шрифтом, а индексы перечисляются через запятую. Например, $A_{1,1}$ – элемент $\boldsymbol{A}$ в левом верхнем углу, а $\boldsymbol{A}_{m,n}$ – в правом нижнем. Для адресации всех элементов в одной строке мы указываем номер этой строки i, а вместо номера столбца – двоеточие, например $\boldsymbol{A}_{i,:}$ обозначает i-ю строку $\boldsymbol{A}$. Аналогично $\boldsymbol{A}_{:,i}$ обозначает i-й столбец $\boldsymbol{A}$. Если требуется явно поименовать элементы матрицы, то мы записываем их в виде прямоугольной таблицы, заключенной в квадратные скобки:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}. \tag{2.2}$$

Иногда требуется индексировать матричные выражения, не обозначенные одной буквой. Тогда нижние индексы ставятся после всего выражения. Например, $f(\boldsymbol{A})_{i,j}$ обозначает элемент в позиции (i, j) матрицы, полученной применением функции f к матрице $\boldsymbol{A}$.

- **Тензоры.** Бывает, что нужен массив размерности, большей 2. В общем случае массив чисел в узлах равномерной сетки с произвольным числом осей называется тензором. Тензор с именем «A» будет обозначаться таким шрифтом: $\mathbf{A}$, а его элемент в позиции (i, j, k) – $\mathsf{A}_{i,j,k}$.

Важной операцией над матрицами является **транспонирование**. Транспонированной называется матрица, отраженная относительно главной диагонали, идущей из левого верхнего угла направо и вниз. На рис. 2.1 показано графическое изображение этой операции. Результат транспонирования матрицы $\boldsymbol{A}$ обозначается $\boldsymbol{A}^\top$ и формально определяется следующим образом:

$$(\boldsymbol{A}^\top)_{i,j} = A_{j,i}. \tag{2.3}$$

Векторы можно считать матрицами с одним столбцом. Тогда результатом транспонирования вектора будет матрица с одной строкой. Иногда мы будем записывать

вектор прямо в основном тексте в виде матрицы-строки, тогда транспонирование преобразует его в стандартный вектор-столбец, например: $\boldsymbol{x} = [x_1, x_2, x_3]^\top$.

Скаляр можно рассматривать как матрицу из одного элемента. Это значит, что результатом транспонирования скаляра является он сам: $a = a^\top$.

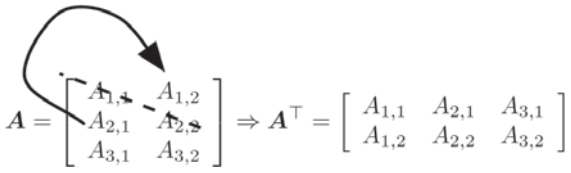

Рис. 2.1 ❖ Транспонирование матрицы можно рассматривать как отражение относительно главной диагонали

Матрицы одинакового размера можно складывать поэлементно: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ означает, что $C_{i,j} = A_{i,j} + B_{i,j}$.

Определены также операции сложения матрицы со скаляром и умножения матрицы на скаляр. Для этого нужно произвести соответствующую операцию для каждого элемента матрицы: $\boldsymbol{D} = a \cdot \boldsymbol{B} + c$ означает, что $D_{i,j} = a \cdot B_{i,j} + c$.

В контексте глубокого обучения применяется также не совсем традиционная нотация. Мы определяем операцию сложения матрицы и вектора, дающую матрицу: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{b}$ означает, что $C_{i,j} = A_{i,j} + b_j$. Иными словами, вектор $\boldsymbol{b}$ прибавляется к каждой строке матрицы. Такая сокращенная запись позволяет обойтись без определения матрицы, в которой каждая строка содержит вектор $\boldsymbol{b}$. Это неявное копирование $\boldsymbol{b}$ сразу в несколько мест называется **укладыванием** (broadcasting).

2.2. Умножение матриц и векторов

Одна из самых важных операций над матрицами – перемножение двух матриц. Произведением матриц $\boldsymbol{A}$ и $\boldsymbol{B}$ также является матрица, $\boldsymbol{C}$. Она определена, только если число столбцов $\boldsymbol{A}$ равно числу строк $\boldsymbol{B}$. Если $\boldsymbol{A}$ имеет размер $m \times n$, а $\boldsymbol{B}$ – размер $n \times p$, то $\boldsymbol{C}$ будет иметь размер $m \times p$. Для обозначения произведения имена матриц записываются подряд, например:

$$\boldsymbol{C} = \boldsymbol{AB}. \tag{2.4}$$

Произведение определяется следующим образом:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}. \tag{2.5}$$

Отметим, что обычное произведение двух матриц получается не в результате **поэлементного перемножения**. Но такая операция тоже существует – она называется **произведением Адамара** и обозначается $\boldsymbol{A} \odot \boldsymbol{B}$.

Скалярным произведением двух векторов $\boldsymbol{x}$ и $\boldsymbol{y}$ одинаковой размерности называется произведение матриц $\boldsymbol{x}^\top \boldsymbol{y}$. Таким образом, элемент $C_{i,j}$ матрицы $\boldsymbol{C} = \boldsymbol{AB}$ является скалярным произведением i-й строки $\boldsymbol{A}$ и j-го столбца $\boldsymbol{B}$.

Операции над матрицами обладают рядом полезных свойств, упрощающих их математический анализ. Например, операция умножения дистрибутивна относительно сложения:

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{AB} + \boldsymbol{AC}. \tag{2.6}$$

Она также ассоциативна:

$$\boldsymbol{A}(\boldsymbol{BC}) = (\boldsymbol{AB})\boldsymbol{C}. \tag{2.7}$$

В отличие от умножения скаляров, умножение матриц не коммутативно (равенство $\boldsymbol{AB} = \boldsymbol{BA}$ справедливо не всегда). Но скалярное произведение векторов коммутативно:

$$\boldsymbol{x}^\top \boldsymbol{y} = \boldsymbol{y}^\top \boldsymbol{x}. \tag{2.8}$$

Операции транспонирования и умножения связаны простой формулой:

$$(\boldsymbol{AB})^\top = \boldsymbol{B}^\top \boldsymbol{A}^\top. \tag{2.9}$$

Отсюда следует доказательство тождества (2.8), поскольку скалярное произведение – это скаляр и, значит, совпадает с результатом транспонирования:

$$\boldsymbol{x}^\top \boldsymbol{y} = (\boldsymbol{x}^\top \boldsymbol{y})^\top = \boldsymbol{y}^\top \boldsymbol{x}. \tag{2.10}$$

Поскольку эта книга – не учебник по линейной алгебре, мы не станем приводить полный список полезных свойств матричного произведения, а просто сообщим, что их еще очень много.

Теперь мы знаем о принятых в линейной алгебре обозначениях достаточно, для того чтобы записать систему линейных уравнений:

$$\boldsymbol{Ax} = \boldsymbol{b}, \tag{2.11}$$

где $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ – известная матрица, $\boldsymbol{b} \in \mathbb{R}^m$ – известный вектор, а $\boldsymbol{x} \in \mathbb{R}^n$ – неизвестный вектор, элементы которого, x_i, мы хотим найти. Каждая строка $\boldsymbol{A}$ и соответственный элемент $\boldsymbol{b}$ дают одно ограничение. Уравнение (2.11) можно переписать в виде:

$$\boldsymbol{A}_{1,:}\boldsymbol{x} = b_1 \tag{2.12}$$
$$\boldsymbol{A}_{2,:}\boldsymbol{x} = b_2 \tag{2.13}$$
$$\ldots \tag{2.14}$$
$$\boldsymbol{A}_{m,:}\boldsymbol{x} = b_m \tag{2.15}$$

или даже в еще более явном виде:

$$\boldsymbol{A}_{1,1}x_1 + \boldsymbol{A}_{1,2}x_2 + \ldots + \boldsymbol{A}_{1,n}x_n = b_1 \tag{2.16}$$
$$\boldsymbol{A}_{2,1}x_1 + \boldsymbol{A}_{2,2}x_2 + \ldots + \boldsymbol{A}_{2,n}x_n = b_2 \tag{2.17}$$
$$\ldots \tag{2.18}$$
$$\boldsymbol{A}_{m,1}x_1 + \boldsymbol{A}_{m,2}x_2 + \ldots + \boldsymbol{A}_{m,n}x_n = b_m \tag{2.19}$$

Нотация умножения матрицы на вектор позволяет записывать такие уравнения более компактно.

2.3. Единичная и обратная матрица

В линейной алгебре определена операция обращения матриц, которая позволяет аналитически решить уравнение (2.11) для многих значений $\boldsymbol{A}$.

Чтобы описать операцию обращения, мы должны сначала определить **единичную матрицу**. Так называется матрица, которая при умножении на любой вектор оставля-

ет этот вектор без изменения. Единичную матрицу, сохраняющую n-мерные векторы, будем обозначать $\boldsymbol{I}_n$. Формально говоря, $\boldsymbol{I}_n \in \mathbb{R}^{n \times n}$ и

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n \boldsymbol{x} = \boldsymbol{x}. \tag{2.20}$$

Единичная матрица устроена просто: все элементы главной диагонали равны единице, а все остальные – нулю. Пример приведен на рис. 2.2.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Рис. 2.2 ❖ Пример единичной матрицы $\boldsymbol{I}_3$

Матрица, обратная к $\boldsymbol{A}$, обозначается $\boldsymbol{A}^{-1}$ и по определению удовлетворяет следующему соотношению:

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n. \tag{2.21}$$

Теперь для решения уравнения (2.11) нужно проделать следующие действия:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}; \tag{2.22}$$
$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}; \tag{2.23}$$
$$\boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}; \tag{2.24}$$
$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}. \tag{2.25}$$

Разумеется, эта процедура предполагает, что матрицу $\boldsymbol{A}^{-1}$ можно найти. В следующем разделе мы обсудим условия существования обратной матрицы.

В случае, когда $\boldsymbol{A}^{-1}$ существует, для ее нахождения в замкнутой форме можно применить один из нескольких алгоритмов. Теоретически один раз найденную обратную матрицу можно использовать многократно для решения уравнения с разными значениями $\boldsymbol{b}$. Однако на практике $\boldsymbol{A}^{-1}$ редко используется в программах. Поскольку матрицу $\boldsymbol{A}^{-1}$ можно представить в компьютере лишь с ограниченной точностью, алгоритмы, в которых используется значение $\boldsymbol{b}$, обычно дают более точные оценки $\boldsymbol{x}$.

2.4. Линейная зависимость и линейная оболочка

Для существования $\boldsymbol{A}^{-1}$ уравнение (2.11) должно иметь единственное решение для любого значения $\boldsymbol{b}$. Бывает и так, что система уравнений не имеет ни одного решения или имеет бесконечно много решений для некоторых значений $\boldsymbol{b}$. Но никогда не может быть так, что система имеет конечное число решений, большее 1; если $\boldsymbol{x}$ и $\boldsymbol{y}$ – решения, то

$$\boldsymbol{z} = \alpha\boldsymbol{x} + (1 - \alpha)\boldsymbol{y} \tag{2.26}$$

тоже решение при любом вещественном α.

Чтобы проанализировать, сколько решений имеет уравнение, представим себе, что столбцы $\boldsymbol{A}$ определяют различные направления от **начала координат** (точки, соответствующей вектору, все элементы которого равны нулю), а затем подумаем, сколько

есть способов достичь точки $\boldsymbol{b}$. Тогда элемент x_i определяет, как далеко следует пройти в направлении столбца i:

$$\boldsymbol{Ax} = \sum_i x_i \boldsymbol{A}_{:,i}. \tag{2.27}$$

В общем случае такое выражение называется **линейной комбинацией**. Формально для получения линейной комбинации некоторого множества векторов $\{\boldsymbol{v}^{(1)}, ..., \boldsymbol{v}^{(n)}\}$ нужно умножить каждый вектор $\boldsymbol{v}^{(i)}$ на скалярный коэффициент и сложить результаты:

$$\sum_i c_i \boldsymbol{v}^{(i)}. \tag{2.28}$$

Линейной оболочкой множества векторов называется множество всех векторов, представимых в виде их линейных комбинаций.

Таким образом, чтобы определить, имеет ли уравнение $\boldsymbol{Ax} = \boldsymbol{b}$ решение, нужно проверить, входит ли $\boldsymbol{b}$ в линейную оболочку столбцов матрицы $\boldsymbol{A}$. Эта линейная оболочка называется **пространством столбцов**, или **областью значений** матрицы $\boldsymbol{A}$.

Итак, чтобы система $\boldsymbol{Ax} = \boldsymbol{b}$ имела решение для любого $\boldsymbol{b} \in \mathbb{R}^m$, необходимо, чтобы пространство столбцов $\boldsymbol{A}$ совпадало с $\mathbb{R}^m$. Если в $\mathbb{R}^m$ существует точка $\boldsymbol{b}$, не принадлежащая пространству столбцов, то для такого значения $\boldsymbol{b}$ система не будет иметь решения. Из требования о совпадении пространства столбцов $\boldsymbol{A}$ с $\mathbb{R}^m$ сразу следует, что в $\boldsymbol{A}$ должно быть по меньшей мере m столбцов, т. е. $n \geq m$. Иначе размерность пространства столбцов была бы меньше m. Рассмотрим, к примеру, матрицу 3×2. Правая часть $\boldsymbol{b}$ имеет размерность 3, а размерность $\boldsymbol{x}$ равна всего 2, поэтому путем изменения $\boldsymbol{x}$ мы в лучшем случае сможем замести двумерную плоскость в $\mathbb{R}^3$. Уравнение будет иметь решение тогда и только тогда, когда $\boldsymbol{b}$ лежит в этой плоскости.

Условие $n \geq m$ необходимо, чтобы система имела решение для любой правой части, но недостаточно, поскольку некоторые столбцы могут быть избыточны. Рассмотрим матрицу 2×2, в которой оба столбца одинаковы. Ее пространство столбцов такое же, как у матрицы 2×1, содержащей только один экземпляр повторяющегося столбца. Иными словами, пространство столбцов – прямая, не покрывающая все $\mathbb{R}^2$, несмотря на то что столбцов два.

Такой вид избыточности называется **линейной зависимостью**. Множество векторов называется **линейно независимым**, если ни один вектор из этого множества не является линейной комбинацией других. Если добавить во множество вектор, являющийся линейной комбинацией каких-то других, то линейная оболочка исходного множества не пополнится новыми точками. Следовательно, для того чтобы пространство столбцов матрицы совпадало со всем пространством $\mathbb{R}^m$, матрица должна содержать, по меньшей мере, один набор m линейно независимых столбцов. Это необходимое и достаточное условие существования у уравнения (2.11) решения для любого значения $\boldsymbol{b}$. Отметим, что в условии говорится о наличии в точности m линейно независимых столбцов в наборе, а не хотя бы m. Ни один набор m-мерных векторов не может содержать более m линейно независимых векторов, но матрица, имеющая более m столбцов, может содержать несколько таких наборов.

Для существования обратной матрицы необходимо еще, чтобы уравнение (2.11) имело не более одного решения для каждого значения $\boldsymbol{b}$. Это означает, что в матрице может быть не больше m столбцов, иначе существовало бы более одного способа параметризовать каждое решение.

Собирая все вместе, мы заключаем, что матрица должна быть квадратной, т. е. $m = n$, и что ее столбцы должны быть линейно независимы. Квадратная матрица с линейно зависимыми столбцами называется **вырожденной** (особенной, сингулярной).

Если $\boldsymbol{A}$ не квадратная или квадратная, но вырожденная, то решить уравнение все-таки можно, но метод обращения матрицы не подойдет.

До сих пор при обсуждении обратных матриц мы имели в виду умножение слева. Но можно также определить матрицу, обратную относительно умножения справа:

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}. \tag{2.29}$$

Для квадратных матриц обе обратные матрицы совпадают.

2.5. Нормы

Иногда требуется получить длину вектора. В машинном обучении для измерения длины применяются функции, называемые **нормами**. Норма L^p определяется следующим образом:

$$\|\boldsymbol{x}\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}. \tag{2.30}$$

для $p \in \mathbb{R}$, $p \geqslant 1$.

Нормы, в т. ч. и норма L^p, – это функции, отображающие векторы на неотрицательные числа. Интуитивно норма вектора $\boldsymbol{x}$ измеряет расстояние от точки $\boldsymbol{x}$ до начала координат. Более строго, нормой называется любая функция f, обладающая следующими свойствами:

- $f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = 0$;
- $f(\boldsymbol{x} + \boldsymbol{y}) \leqslant f(\boldsymbol{x}) + f(\boldsymbol{y})$ (**неравенство треугольника**);
- $\forall \alpha \in \mathbb{R}\, f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$.

Норма L^2 называется **евклидовой нормой**, это просто евклидово расстояние от начала координат до точки, определяемой вектором $\boldsymbol{x}$. Норма L^2 применяется в машинном обучении так часто, что ее обозначают просто $\|\boldsymbol{x}\|$, опуская надстрочный индекс 2. Кроме того, длину вектора принято измерять как квадрат нормы L^2, который можно записать в виде $\boldsymbol{x}^\top \boldsymbol{x}$.

Квадрат нормы L^2 удобнее самой нормы с точки зрения математических выкладок и вычислений. Например, частная производная квадрата нормы L^2 по каждому элементу $\boldsymbol{x}$ зависит только от этого элемента, тогда как производные самой нормы L^2 зависят от всего вектора в целом. Во многих контекстах квадрат нормы L^2 нежелателен, потому что он слишком медленно растет вблизи начала координат. В некоторых приложениях машинного обучения важно различать элементы, в точности равные нулю и мало отличающиеся от нуля. В таких случаях мы прибегаем к функции, которая всюду растет с одинаковой скоростью, но сохраняет математическую простоту: норме L^1. Норму L^1 можно записать в виде:

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i|. \tag{2.31}$$

Эта норма часто используется в машинном обучении, когда важно различать нулевые и ненулевые элементы. Всякий раз как элемент вектора $\boldsymbol{x}$ отдаляется от 0 на ε, норма L^1 увеличивается на ε.

Иногда длина вектора измеряется количеством его ненулевых элементов. Некоторые авторы называют такую функцию «нормой L^0», но такая терминология некорректна. Количество ненулевых элементов вектора нормой не является, поскольку при умножении вектора на α это количество не меняется. Зачастую вместо подсчета ненулевых элементов используют норму L^1.

В машинном обучении также часто возникает норма L^∞, которую еще называют **максимальной нормой**. Она определяется как максимальное абсолютное значение элементов вектора:

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i|. \tag{2.32}$$

Иногда возникает необходимость оценить размер матрицы. В контексте глубокого обучения для этого обычно применяют **норму Фробениуса**:

$$\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}, \tag{2.33}$$

аналогичную норме L^2 вектора.

Скалярное произведение двух векторов можно выразить через нормы:

$$\boldsymbol{x}^\top \boldsymbol{y} = \|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2 \cos\theta, \tag{2.34}$$

где θ – угол между $\boldsymbol{x}$ и $\boldsymbol{y}$.

2.6. Специальные виды матриц и векторов

Некоторые частные случаи матриц и векторов особенно полезны.

В **диагональной матрице** все элементы, кроме находящихся на главной диагонали, равны нулю. Точнее, матрица $\boldsymbol{D}$ называется диагональной, если $D_{i,j} = 0$ для всех $i \neq j$. Примером диагональной матрицы служит единичная матрица, в которой все элементы на главной диагонали равны 1. Квадратная диагональная матрица, диагональ которой описывается вектором $\boldsymbol{v}$, обозначается $\text{diag}(\boldsymbol{v})$. Диагональные матрицы представляют особый интерес, потому что произведение с такой матрицей вычисляется очень эффективно. Чтобы вычислить $\text{diag}(\boldsymbol{v})\boldsymbol{x}$, нужно умножить каждый элемент x_i на v_i. Иначе говоря, $\text{diag}(\boldsymbol{v})\boldsymbol{x} = \boldsymbol{v} \odot \boldsymbol{x}$. Обратить квадратную диагональную матрицу тоже просто. Обратная к ней матрица существует, только если все диагональные элементы отличны от нуля, и тогда $\text{diag}(\boldsymbol{v})^{-1} = \text{diag}([1/v_1, ..., 1/v_n]^\top)$. Во многих случаях алгоритм машинного обучения можно сформулировать в терминах произвольных матриц, но получить менее накладный (но и менее общий) алгоритм, ограничившись только диагональными матрицами.

Диагональная матрица может быть не только квадратной, но и прямоугольной. У неквадратной диагональной матрицы нет обратной, но умножение на нее все равно обходится дешево. Произведение диагональной матрицы $\boldsymbol{D}$ на вектор $\boldsymbol{x}$ сводится к умножению каждого элемента $\boldsymbol{x}$ на некоторое число и дописыванию нулей в конец получившегося вектора, если высота $\boldsymbol{D}$ больше ее ширины, или отбрасыванию последних элементов вектора в противном случае.

Матрица называется **симметричной**, если совпадает с транспонированной:

$$\boldsymbol{A} = \boldsymbol{A}^\top. \tag{2.35}$$

Симметричные матрицы часто образуются, когда их элементы порождаются некоторой функцией двух аргументов, результат которой не зависит от порядка аргументов. Например, если $A_{i,j}$ – расстояние от точки i до точки j, то $A_{i,j} = A_{j,i}$, поскольку функция расстояния симметрична.

Единичным вектором называется вектор с нормой, равной 1:

$$\|\boldsymbol{x}\|_2 = 1. \tag{2.36}$$

Говорят, что векторы $\boldsymbol{x}$ и $\boldsymbol{y}$ ортогональны, если $\boldsymbol{x}^\top \boldsymbol{y} = 0$. Если нормы обоих векторов не равны 0, то это значит, что угол между векторами составляет 90 градусов. В пространстве $\mathbb{R}^n$ можно найти не более n взаимно ортогональных ненулевых векторов. Ортогональные векторы, норма которых равна 1, называются **ортонормированными**.

Ортогональной называется квадратная матрица, строки и столбцы которой образуют ортонормированные системы векторов:

$$\boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{I}. \tag{2.37}$$

Отсюда следует, что

$$\boldsymbol{A}^{-1} = \boldsymbol{A}^\top. \tag{2.38}$$

Ортогональные матрицы интересны из-за простоты вычисления обратной матрицы. Обратите внимание на определение ортогональной матрицы: ее строки должны образовывать не просто ортогональную, а ортонормированную систему векторов. Не существует специального термина для матриц, строки и столбцы которых образуют ортогональную, но не ортонормированную систему векторов.

2.7. Спектральное разложение матрицы

Многие математические объекты удается лучше понять, если разложить их на составные части или найти какие-то универсальные свойства, не зависящие от способа представления.

Например, целые числа можно разложить на простые множители. Способ представления числа 12 зависит от того, в какой системе счисления оно записано, например двоичной или десятичной, но в любом случае $12 = 2 \times 2 \times 3$. Из этого представления можно вывести ряд полезных свойств, например что 12 не делится на 5 или что любое число, делящееся на 12, делится также на 3.

Таким образом, истинная природа целого числа раскрывается в результате разложения его на простые множители. И точно так же мы можем разложить матрицу и тем самым получить о ее функциональных свойствах некоторую информацию, не очевидную из представления матрицы в виде массива элементов.

Одно из самых популярных разложений матрицы называется **спектральным разложением**, или разложением на множество собственных векторов и собственных значений.

Собственным вектором квадратной матрицы $\boldsymbol{A}$ называется ненулевой вектор $\boldsymbol{v}$ – такой, что умножение $\boldsymbol{A}$ на $\boldsymbol{v}$ изменяет лишь масштаб $\boldsymbol{v}$:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}. \tag{2.39}$$

Скаляр λ называется **собственным значением**, соответствующим этому собственному вектору. (Можно также искать левые собственные векторы, для которых $\boldsymbol{v}^\top \boldsymbol{A} = \lambda \boldsymbol{v}^\top$, но обычно нас интересуют только правые собственные векторы.)

Если $\boldsymbol{v}$ – собственный вектор $\boldsymbol{A}$, то собственным будет и вектор $s\boldsymbol{v}$ для любого $s \in \mathbb{R}$, $s \neq 0$. Более того, вектору $s\boldsymbol{v}$ соответствует то же собственное значение, что и $\boldsymbol{v}$. Поэтому мы обычно ищем только единичные собственные векторы.

Пусть матрица $\boldsymbol{A}$ имеет n линейно независимых собственных векторов $\{v^{(1)}, ..., v^{(n)}\}$ с собственными значениями $\{\lambda_1, ..., \lambda_n\}$. Образуем из них матрицу $\boldsymbol{V}$, в которой каждый столбец – это собственный вектор: $\boldsymbol{V} = [v^{(1)}, ..., v^{(n)}]$. А из собственных значений образуем вектор $\lambda = [\lambda_1, ..., \lambda_n]^\top$. Тогда **спектральное разложение** матрицы A описывается формулой:

$$\boldsymbol{A} = \boldsymbol{V}\,\mathrm{diag}(\lambda)\,\boldsymbol{V}^{-1}. \tag{2.40}$$

Мы видели, что конструирование матриц с заданными собственными значениями и собственными векторами позволяет растягивать пространство в нужных направлениях. Но часто бывает нужно разложить имеющуюся матрицу по ее собственным векторам и собственным значениям. Это помогает анализировать некоторые свойства матрицы точно так же, как разложение целого числа на простые множители помогает понять поведение этого числа.

Не у каждой матрицы есть спектральное разложение. Иногда спектральное разложение существует, но состоит из комплексных, а не вещественных чисел. К счастью, в этой книге нам обычно придется иметь дело только с матрицами специального вида, у которых имеется простое разложение. Точнее, у любой симметричной вещественной матрицы все собственные векторы и собственные значения вещественные.

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top, \tag{2.41}$$

где $\boldsymbol{Q}$ – ортогональная матрица, образованная собственными векторами $\boldsymbol{A}$, а $\boldsymbol{\Lambda}$ – диагональная матрица. Собственное значение $\boldsymbol{\Lambda}_{i,i}$ ассоциировано с собственным вектором в i-м столбце $\boldsymbol{Q}$, обозначаемым $Q_{:,i}$. Поскольку $\boldsymbol{Q}$ – ортогональная матрица, можно считать, что $\boldsymbol{A}$ масштабирует пространство с коэффициентом λ_i в направлении $v^{(i)}$. На рис. 2.3 показан пример.

Хотя для любой симметричной вещественной матрицы $\boldsymbol{A}$ существует спектральное разложение, это разложение может быть не единственным. Если какие-то два или более собственных векторов имеют одинаковое собственное значение, то любые ортогональные векторы, принадлежащие их линейной оболочке, также будут собственными векторами $\boldsymbol{A}$ с тем же самым собственным значением, поэтому матрицу $\boldsymbol{Q}$ можно с тем же успехом образовать из этих векторов. По принятому соглашению элементы $\boldsymbol{\Lambda}$ обычно располагаются в порядке убывания. При таком соглашении спектральное разложение единственно, только если все собственные значения различны.

Спектральное разложение может многое рассказать о матрице. Матрица является вырожденной тогда и только тогда, когда у нее есть хотя бы одно нулевое собственное значение. Кроме того, спектральное разложение вещественной симметричной матрицы можно использовать для оптимизации квадратичных форм вида $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}$ при условии $\|\boldsymbol{x}\|_2 = 1$. Если $\boldsymbol{x}$ – собственный вектор $\boldsymbol{A}$, то f равно его собственному значению. Максимальное (минимальное) значение f при заданном ограничении равно максимальному (минимальному) собственному значению.

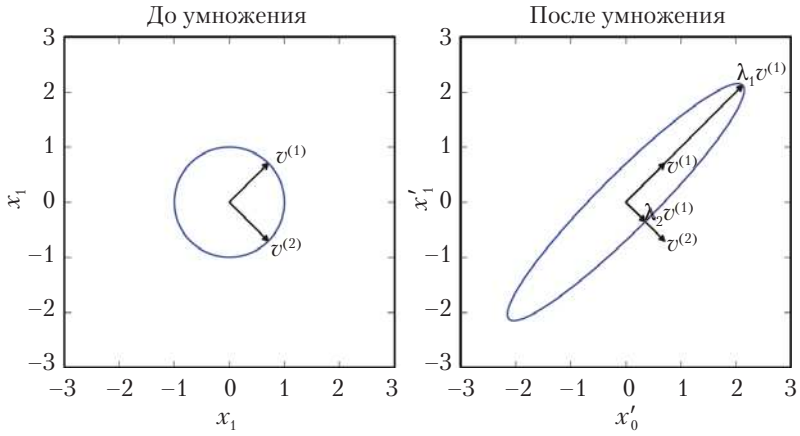

Рис. 2.3 ❖ Геометрический смысл собственных векторов и собственных значений. У матрицы $\boldsymbol{A}$ два ортонормированных собственных вектора: $\boldsymbol{v}^{(1)}$ с собственным значением λ_1 и $\boldsymbol{v}^{(2)}$ с собственным значением λ_2. (*Слева*) Множество всех единичных векторов $\boldsymbol{u} \in \mathrm{R}2$ изображено в виде единичной окружности. (*Справа*) Изображено множество точек $\boldsymbol{Au}$. Наблюдая, во что $\boldsymbol{A}$ переводит единичную окружность, мы можем сделать вывод, что она масштабирует пространство в направлении $\boldsymbol{v}^{(i)}$ с коэффициентом λ_i

Матрица, все собственные значения которой положительны, называется **положительно определенной**. Если же все собственные значения положительны или равны нулю, то матрица называется **положительно полуопределенной**. Аналогично, если все собственные значения отрицательны, то матрица называется **отрицательно определенной**, а если отрицательны или равны нулю – то **отрицательно полуопределенной**. Положительно полуопределенные матрицы интересны тем, что для них выполняется неравенство $\boldsymbol{x}^\top \boldsymbol{Ax} \geq 0$ при любом $\boldsymbol{x}$. Положительная определенность дополнительно гарантирует, что $\boldsymbol{x}^\top \boldsymbol{Ax} = 0 \Rightarrow \boldsymbol{x} = 0$.

2.8. Сингулярное разложение

В разделе 2.7 мы видели, как разложить матрицу по собственным векторам и собственным значениям. **Сингулярное разложение** (singular value decomposition —SVD) – это другой способ разложения матрицы: по **сингулярным векторам** и **сингулярным значениям**. SVD несет ту же информацию, что спектральное разложение, но применимо в более общем случае. Сингулярное разложение есть у любой вещественной матрицы, чего не скажешь о спектральном разложении. Например, если матрица не квадратная, то спектральное разложение не определено, поэтому мы вынуждены использовать сингулярное разложение.

Напомним, что для получения спектрального разложения матрицы $\boldsymbol{A}$ нужно найти матрицу $\boldsymbol{V}$ собственных векторов и вектор $\boldsymbol{\lambda}$ собственных значений – такие, что:

$$\boldsymbol{A} = \boldsymbol{V}\,\mathrm{diag}(\boldsymbol{\lambda})\,\boldsymbol{V}^{-1}. \tag{2.42}$$

Сингулярное разложение аналогично, но теперь $\boldsymbol{A}$ записывается в виде произведения трех матриц:

$$\boldsymbol{A} = \boldsymbol{UDV}^{\top}. \tag{2.43}$$

Пусть $\boldsymbol{A}$ – матрица $m \times n$. Тогда $\boldsymbol{U}$ должна быть матрицей $m \times m$, $\boldsymbol{D}$ – матрицей $m \times n$, а $\boldsymbol{V}$ – матрицей $n \times n$.

Эти матрицы обладают определенными свойствами. Матрицы $\boldsymbol{U}$ и $\boldsymbol{V}$ ортогональные, а матрица $\boldsymbol{D}$ диагональная (при этом необязательно квадратная).

Элементы на диагонали $\boldsymbol{D}$ называются **сингулярными значениями** матрицы $\boldsymbol{A}$. Столбцы $\boldsymbol{U}$ называются **левыми сингулярными векторами**, а столбцы $\boldsymbol{V}$ – **правыми сингулярными векторами**.

Сингулярное разложение $\boldsymbol{A}$ можно интерпретировать в терминах спектрального разложения функций $\boldsymbol{A}$. Левые сингулярные векторы $\boldsymbol{A}$ являются собственными векторами $\boldsymbol{AA}^{\top}$, а правые сингулярные векторы $\boldsymbol{A}$ – собственными векторами $\boldsymbol{A}^{\top}\boldsymbol{A}$. Ненулевые сингулярные значения $\boldsymbol{A}$ равны квадратным корням из собственных значений $\boldsymbol{A}^{\top}\boldsymbol{A}$ (и $\boldsymbol{AA}^{\top}$).

Но, пожалуй, самое полезное свойство сингулярного разложения – использование его для обобщения операции обращения матриц на неквадратные матрицы, о чем мы и поговорим в следующем разделе.

2.9. Псевдообратная матрица Мура–Пенроуза

Операция обращения определена только для квадратных матриц. Допустим, что требуется найти левую обратную матрицу $\boldsymbol{B}$ для матрицы $\boldsymbol{A}$, что позволило бы решить линейное уравнение

$$\boldsymbol{Ax} = \boldsymbol{y} \tag{2.44}$$

путем умножения обеих частей слева на $\boldsymbol{B}$:

$$\boldsymbol{x} = \boldsymbol{By} \tag{2.45}$$

Единственное отображение $\boldsymbol{A}$ на $\boldsymbol{B}$ возможно не всегда; это зависит от характера задачи.

Если высота $\boldsymbol{A}$ больше ширины, то у такого уравнения может не оказаться решений. Если же ширина $\boldsymbol{A}$ больше высоты, то решений может быть много.

Операция **псевдообращения Мура–Пенроуза** позволяет кое-что сделать и в этих случаях. Псевдообратная к $\boldsymbol{A}$ матрица определяется следующим образом:

$$\boldsymbol{A}^{+} = \lim_{\alpha \searrow 0}(\boldsymbol{A}^{\top}\boldsymbol{A} + \alpha \boldsymbol{I})^{-1}\boldsymbol{A}^{\top}. \tag{2.46}$$

Применяемые на практике алгоритмы вычисления псевдообратной матрицы основаны не на этом определении, а на формуле

$$\boldsymbol{A}^{+} = \boldsymbol{VD}^{+}\boldsymbol{U}^{\top}, \tag{2.47}$$

где $\boldsymbol{U}$, $\boldsymbol{D}$ и $\boldsymbol{V}$ составляют сингулярное разложение $\boldsymbol{A}$, а псевдообратная матрица $\boldsymbol{D}^{+}$ диагональной матрицы $\boldsymbol{D}$ получается путем обращения всех ненулевых диагональных элементов с последующим транспонированием.

Если число столбцов $\boldsymbol{A}$ больше числа строк, то решение линейного уравнения, найденное псевдообращением, – лишь одно из многих возможных. Точнее, будет найдено решение $\boldsymbol{x} = \boldsymbol{A}^{+}\boldsymbol{y}$ с минимальной евклидовой нормой $\|\boldsymbol{x}\|_2$.

Если число строк $\boldsymbol{A}$ больше числа столбцов, то может не оказаться ни одного решения. В таком случае псевдообращение находит такой вектор $\boldsymbol{x}$, для которого $\boldsymbol{Ax}$ максимально близко к $\boldsymbol{y}$ в терминах евклидовой нормы $\|\boldsymbol{Ax} - \boldsymbol{y}\|_2$.

2.10. Оператор следа

Оператор следа вычисляет сумму всех диагональных элементов матрицы:

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_i \boldsymbol{A}_{i,i}. \tag{2.48}$$

Этот оператор полезен по многим причинам. Некоторые операции, которые трудно выразить, не прибегая к нотации суммирования, можно записать с помощью произведения матриц и оператора следа. Вот, например, как можно переписать определение нормы Фробениуса матрицы:

$$\|A\|_F = \sqrt{Tr(\boldsymbol{A}\boldsymbol{A}^\top)}. \tag{2.49}$$

Если в выражение входит оператор следа, то открываются возможности преобразовывать это выражение, пользуясь различными тождествами. Например, след инвариантен относительно транспонирования:

$$\mathrm{Tr}(\boldsymbol{A}) = \mathrm{Tr}(\boldsymbol{A}^\top). \tag{2.50}$$

След квадратной матрицы, представленной в виде произведения сомножителей, инвариантен также относительно перемещения последнего сомножителя в начало, если формы матриц допускают такую операцию:

$$\mathrm{Tr}(\boldsymbol{ABC}) = \mathrm{Tr}(\boldsymbol{CAB}) = \mathrm{Tr}(\boldsymbol{BCA}), \tag{2.51}$$

или в более общем виде:

$$\mathrm{Tr}\left(\prod_{i=1}^{n} \boldsymbol{F}^{(i)}\right) = \mathrm{Tr}\left(\boldsymbol{F}^{(n)} \prod_{i=1}^{n-1} \boldsymbol{F}^{(i)}\right). \tag{2.52}$$

Эта инвариантность относительно циклической перестановки имеет место даже тогда, когда меняется форма произведения. Например, если $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{B} \in \mathbb{R}^{n \times m}$, то

$$\mathrm{Tr}(\boldsymbol{AB}) = \mathrm{Tr}(\boldsymbol{BA}), \tag{2.53}$$

несмотря на то что $\boldsymbol{AB} \in \mathbb{R}^{m \times m}$, а $\boldsymbol{BA} \in \mathbb{R}^{n \times n}$.

Полезно также помнить, что след скаляра равен ему самому: $a = \mathrm{Tr}(a)$.

2.11. Определитель

Определитель квадратной матрицы, обозначаемый $\det(\boldsymbol{A})$, – это функция, сопоставляющая матрице вещественное число. Определитель равен произведению всех собственных значений матрицы. Абсолютную величину определителя можно рассматривать как меру сжатия или расширения пространства матрицей. Если определитель равен 0, то пространство полностью сворачивается хотя бы по одному измерению, т. е. теряется весь объем. Если определитель равен 1, то преобразование сохраняет объем.

2.12. Пример: метод главных компонент

Один из простых алгоритмов машинного обучения, метод главных компонент (principal components analysis – PCA), можно вывести из основ линейной алгебры.

Пусть имеется набор m точек $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ в пространстве $\mathbb{R}^n$, и мы хотим подвергнуть их сжатию с потерей информации, т. е. сохранить точки в меньшем объеме памяти, возможно, ценой некоторой потери точности. Но хотелось бы свести эти потери к минимуму.

Один из способов кодирования точек – представить их в пространстве меньшей размерности. Для каждой точки $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ мы ищем соответствующий ей кодовый вектор $\boldsymbol{c}^{(i)} \in \mathbb{R}^l$. Если l меньше n, то для хранения кодированных точек потребуется меньше памяти, чем для исходных. Мы хотим найти функцию кодирования $f(\boldsymbol{x}) = \boldsymbol{c}$ и функцию декодирования, реконструирующую исходную точку по кодированной, $\boldsymbol{x} \approx g(f(\boldsymbol{x}))$.

Алгоритм PCA определяется выбором функции декодирования. Чтобы упростить декодер, мы будем использовать умножение матриц для обратного перехода в $\mathbb{R}^n$. Пусть $g(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}$, где $\boldsymbol{D} \in \mathbb{R}^{n \times l}$ – матрица, определяющая декодирование.

Вычисление оптимального декодера может оказаться трудной задачей. Для ее упрощения в методе PCA на матрицу $\boldsymbol{D}$ налагается ограничение: ее столбцы должны быть ортогональны (но это не означает, что $\boldsymbol{D}$ – ортогональная матрица в определенном выше смысле, разве что $l = n$).

У поставленной задачи бесконечно много решений, поскольку мы можем увеличить масштаб $D_{:,i}$, одновременно пропорционально уменьшив c_i для всех точек. Для получения единственного решения потребуем еще, чтобы норма всех столбцов $\boldsymbol{D}$ была равна 1.

Чтобы превратить смутную идею в алгоритм, нужно первым делом решить, как генерировать оптимальную кодовую точку $\boldsymbol{c}^*$ для каждой исходной точки $\boldsymbol{x}$. Один из способов – минимизировать расстояние между $\boldsymbol{x}$ и ее реконструкцией $g(\boldsymbol{c}^*)$. Для измерения этого расстояния применим какую-нибудь норму. В алгоритме главных компонент берется норма L^2:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} \|\boldsymbol{x} - g(\boldsymbol{c})\|_2. \tag{2.54}$$

Мы можем использовать квадрат нормы L^2, а не ее саму, потому что минимум достигается при одном и том же значении $\boldsymbol{c}$, т. к. норма L^2 неотрицательна, а операция возведения в квадрат – монотонно возрастающая для неотрицательных аргументов.

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} \|\boldsymbol{x} - g(\boldsymbol{c})\|_2^2. \tag{2.55}$$

Минимизируемую функцию можно переписать в виде:

$$(\boldsymbol{x} - g(\boldsymbol{c}))^\top (\boldsymbol{x} - g(\boldsymbol{c})) \tag{2.56}$$

(по определению нормы L^2 из формулы (2.30))

$$= \boldsymbol{x}^\top \boldsymbol{x} - \boldsymbol{x}^\top g(\boldsymbol{c}) - g(\boldsymbol{c})^\top \boldsymbol{x} + g(\boldsymbol{c})^\top g(\boldsymbol{c}) \tag{2.57}$$

(в силу дистрибутивности)

$$= \boldsymbol{x}^\top \boldsymbol{x} - 2\boldsymbol{x}^\top g(\boldsymbol{c}) + g(\boldsymbol{c})^\top g(c) \tag{2.58}$$

(поскольку результат транспонирования скаляра $g(\boldsymbol{c})^\top \boldsymbol{x}$ совпадает с ним самим).

Снова изменим минимизируемую функцию, опустив первый член, не зависящий от $\boldsymbol{c}$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top g(\boldsymbol{c}) + g(\boldsymbol{c})^\top g(c). \tag{2.59}$$

Теперь подставим сюда определение $g(\boldsymbol{c})$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{D}^\top \boldsymbol{D}c \tag{2.60}$$

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{I}_l c \tag{2.61}$$

(в силу ограничений на ортогональность и нормированность $\boldsymbol{D}$)

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}. \tag{2.62}$$

Для решения этой задачи оптимизации мы можем воспользоваться векторным математическим анализом (если вы не знаете, что это такое, обратитесь к разделу 4.3):

$$\nabla_c(-2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}) = 0 \tag{2.63}$$

$$-2\boldsymbol{D}^\top \boldsymbol{x} + 2\boldsymbol{c} = 0 \tag{2.64}$$

$$\boldsymbol{c} = \boldsymbol{D}^\top \boldsymbol{x}. \tag{2.65}$$

Получается очень эффективный алгоритм: для оптимального кодирования $\boldsymbol{x}$ достаточно всего лишь операций над матрицами и векторами, т. е. функция кодирования выглядит так:

$$f(\boldsymbol{x}) = \boldsymbol{D}^\top \boldsymbol{x}. \tag{2.66}$$

Еще раз применив умножение на матрицу, мы сможем определить операцию реконструкции в алгоритме PCA:

$$r(\boldsymbol{x}) = g(f(\boldsymbol{x})) = \boldsymbol{D}\boldsymbol{D}^\top \boldsymbol{x}. \tag{2.67}$$

Далее следует выбрать кодировочную матрицу $\boldsymbol{D}$. Для этого вернемся к идее минимизации расстояния в смысле нормы L^2 между исходными и реконструированными точками. Поскольку для декодирования всех точек используется одна и та же матрица $\boldsymbol{D}$, мы больше не можем рассматривать точки изолированно, а должны минимизировать норму Фробениуса матрицы ошибок, вычисленных для всех измерений и точек:

$$\boldsymbol{D}^* = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j} (x_j^{(i)} - r(\boldsymbol{x}^{(i)})_j)^2} \text{ при условии } \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l. \tag{2.68}$$

Для вывода алгоритма нахождения $\boldsymbol{D}^*$ сначала рассмотрим случай $l = 1$. Тогда $\boldsymbol{D}$ – это просто одиночный вектор, который мы обозначим $\boldsymbol{d}$. Подставляя (2.67) в (2.68) и заменяя $\boldsymbol{D}$ на $\boldsymbol{d}$, мы сводим задачу к такой:

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i \left\| \boldsymbol{x}^{(i)} - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^{(i)} \right\|_2^2 \text{ при условии } \|\boldsymbol{d}\|_2 = 1. \tag{2.69}$$

Этот прямой результат подстановки – не самый красивый способ записи уравнения. Скалярное значение $\boldsymbol{d}^\top \boldsymbol{x}^{(i)}$ находится справа от вектора $\boldsymbol{d}$. Но обычно скалярные коэффициенты записываются слева от вектора, поэтому перепишем выражение в таком виде:

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i \|\boldsymbol{x}^{(i)} - \boldsymbol{d}^\top \boldsymbol{x}^{(i)} \boldsymbol{d}\|_2^2 \text{ при условии } \|\boldsymbol{d}\|_2 = 1. \tag{2.70}$$

или, воспользовавшись тем, что операция транспонирования не изменяет скаляр:

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(i)\top} \boldsymbol{d}\boldsymbol{d}\|_2^2 \text{ при условии } \|\boldsymbol{d}\|_2 = 1. \tag{2.71}$$

Читателю следует стремиться к свободному владению такими косметическими преобразованиями.

Теперь было бы полезно переформулировать задачу в терминах одной матрицы примеров, а не суммы по отдельным векторам. Это позволит записать ее более компактно. Обозначим $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ матрицу, образованную всеми векторами, рассматриваемыми как строки, т. е. $\boldsymbol{X}_{i,:} = \boldsymbol{x}^{(i)\top}$. Тогда уравнение (2.71) можно переписать в виде:

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \|\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\|_F^2 \text{ при условии } \boldsymbol{d}^\top \boldsymbol{d} = 1. \tag{2.72}$$

Забудем ненадолго об ограничении и упростим часть, содержащую норму Фробениуса:

$$\arg\min_{\boldsymbol{d}} \|\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\|_F^2 \tag{2.73}$$

$$= \arg\min_{\boldsymbol{d}} \mathrm{Tr}((\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top)^\top (\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top)) \tag{2.74}$$

(в силу формулы (2.49))

$$= \arg\min_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X} - \boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top - \boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X} + \boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.75}$$

$$= \arg\min_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}) - \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) - \mathrm{Tr}(\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}) + \mathrm{Tr}(\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.76}$$

$$= \arg\min_{\boldsymbol{d}} - \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) - \mathrm{Tr}(\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}) + \mathrm{Tr}(\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.77}$$

(поскольку члены, не содержащие $\boldsymbol{d}$, на влияют на arg min)

$$= \arg\min_{\boldsymbol{d}} - 2\mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.78}$$

(поскольку матрицы внутри оператора следа можно циклически переставлять, формула (2.52))

$$= \arg\min_{\boldsymbol{d}} - 2\mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.79}$$

(еще раз применяя то же свойство).

Теперь снова вспомним об ограничении:

$$\arg\min_{\boldsymbol{d}} - 2\mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\boldsymbol{d}\boldsymbol{d}^\top) \text{ при условии } \boldsymbol{d}^\top\boldsymbol{d} = 1 \tag{2.80}$$

$$= \arg\min_{\boldsymbol{d}} - 2\mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ при условии } \boldsymbol{d}^\top\boldsymbol{d} = 1 \tag{2.81}$$

(в силу ограничения)

$$= \arg\min_{\boldsymbol{d}} - \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ при условии } \boldsymbol{d}^\top\boldsymbol{d} = 1 \tag{2.82}$$

$$= \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ при условии } \boldsymbol{d}^\top\boldsymbol{d} = 1 \tag{2.83}$$

$$= \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{d}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{d}) \text{ при условии } \boldsymbol{d}^\top\boldsymbol{d} = 1 \tag{2.84}$$

Эту задачу оптимизации можно решить с помощью спектрального разложения: оптимальным будет собственный вектор матрицы $\boldsymbol{X}^\top\boldsymbol{X}$ с наибольшим собственным значением.

Этот вывод применим только к случаю $l = 1$ и восстанавливает лишь первую главную компоненту. В более общем случае, когда требуется восстановить базис главных компонент, матрица $\boldsymbol{D}$ определяется l собственными векторами с наибольшими собственными значениями. Доказательство можно провести по индукции, и мы оставляем его читателю в качестве упражнения.

Линейная алгебра – один из важнейших разделов математики, необходимых для понимания машинного обучения. Другой раздел, без которого машинное обучение немыслимо, – теория вероятностей, к которой мы и переходим.

Глава 3

Теория вероятностей и теория информации

В этой главе мы изложим основы теории вероятностей и теории информации.

Теория вероятностей – это раздел математики, в котором рассматриваются недостоверные утверждения. Она дает средства количественно описать недостоверность, а также аксиомы для вывода новых недостоверных утверждений. В применении к искусственному интеллекту теория вероятностей используется в основном двумя способами. Во-первых, ее правила говорят нам, как должна рассуждать система ИИ, поэтому мы проектируем алгоритмы, которые вычисляют или аппроксимируют различные выражения, полученные на основе теории вероятностей. Во-вторых, теорию вероятностей и математическую статистику можно применить для теоретического анализа поведения предлагаемых систем ИИ.

Теория вероятностей лежит в основе многих научно-технических дисциплин. Эта глава включена для того, чтобы читатели с подготовкой в области программной инженерии, плохо знакомые с теорией вероятностей, понимали изложенный в книге материал.

Если теория вероятностей позволяет формулировать недостоверные утверждения и рассуждать в условиях неопределенности, то теория информации дает возможность количественно оценить меру неопределенности распределения вероятности.

Если вы уже знакомы с теорией вероятностей и теорией информации, то можете сразу перейти к разделу 3.14, в котором описываются графы, применяемые для описания структурных вероятностных моделей в машинном обучении. Если же вы совсем ничего не знаете об этих темах, то изложенного в этой главе материала должно хватить для успешного выполнения исследовательских проектов в области глубокого машинного обучения, но мы все же рекомендуем почитать дополнительные источники, например книгу Jaynes (2003).

3.1. Зачем нужна вероятность?

Во многих разделах информатики имеют дело в основном с детерминированными сущностями. Обычно программист может предполагать, что процессор выполняет машинные команды без ошибок. Аппаратные ошибки бывают, но они настолько редки, что в большинстве программ учитывать такую возможность необязательно. Однако если большинство теоретиков и инженеров-программистов работант в сравнительно стерильных и определенных условиях, то почему же в машинном обучении так часто используется теория вероятностей?

Машинное обучение по необходимости имеет дело с недостоверными, а иногда и стохастическими (недетерминированными) величинами. Недостоверность и недетерминированность проистекают из многих источников. Теоретические аргументы в пользу количественной оценки недостоверности с помощью теории вероятностей приводились уже в 1980-х годах. Многие перечисленные ниже аргументы взяты из работы Pearl (1988) или навеяны ей.

Почти во всех отраслях знания требуется возможность рассуждать в присутствии неопределенности. На самом деле, если не считать математических утверждений, которые истинны по определению, трудно привести пример какого-нибудь высказывания, которое было бы абсолютно верным, или события, которое произойдет гарантированно.

Существуют три источника неопределенности:

1) стохастичность, присущая моделируемой системе. Например, в большинстве интерпретаций квантовой механики динамика субатомных частиц описывается в вероятностных терминах. Можно также сконструировать теоретические сценарии с постулированной случайной динамикой, например гипотетическая карточная игра в предположении, что карты перетасованы случайно;
2) неполнота наблюдаемых данных. Даже детерминированная система может казаться стохастической, если мы не в состоянии наблюдать все переменные, описывающие ее поведение. Например, в парадоксе Монти Холла участник игрового шоу выбирает одну из трех дверей и получает скрытый за ней приз. За двумя дверьми находятся козы, за третьей – автомобиль. Исход при любом выборе участника детерминирован, но, с точки зрения самого участника, исход неопределенный;
3) неполнота модели. Если используется модель, которая отбрасывает часть наблюдаемой информации, то отброшенная информация приводит к недостоверности полученных от модели предсказаний. Допустим, к примеру, что мы конструируем робота, который способен точно фиксировать положения всех находящихся поблизости от него объектов. Если робот дискретизирует пространство, стремясь спрогнозировать положения объектов в будущем, то сам акт дискретизации уже делает информацию о положении объектов недостоверной: каждый объект может находиться в дискретной области, окружающей занимаемое им место в пространстве.

Во многих случаях практичнее использовать простое неопределенное правило, чем сложное определенное, даже если истинное правило детерминировано, и система моделирования позволяет его адекватно представить. Например, простое правило «Большинство птиц умеет летать» легко формулируется и в общем случае полезно, тогда как правило «Птицы умеют летать, за исключением очень молодых птенцов, которые еще не научились летать, больных и травмированных птиц, которые утратили способность летать, нелетающих птиц, включая казуара, страуса и киви...» трудно сформулировать, сопровождать и передавать другим людям, и даже после всех усилий оно оказывается хрупким и уязвимым к неточностям.

Хотя понятно, что необходимы средства для представления недостоверности и рассуждений в условиях неопределенности, не сразу очевидно, что теория вероятностей располагает всеми инструментами, которые нужны в приложениях искусственного интеллекта. Первоначально теория вероятностей разрабатывалась для анализа частоты событий. Легко видеть, как применить ее для изучения таких событий, как сдача карт в покере. Подобные события зачастую повторяемы. Говоря, что вероятность некоторого исхода равна p, мы имеем в виду, что если повторить эксперимент (напри-

мер, сдачу карт) бесконечное число раз, то доля таких исходов составит p. Не понятно, как такое рассуждение может быть применимо к неповторяемым событиям. Когда врач осматривает пациента и говорит, что у него грипп с вероятностью 40 процентов, то это означает что-то совсем другое – мы не можем создать бесконечно много копий пациента и не имеем никаких оснований утверждать, что у разных копий будут такие же симптомы, но различные заболевания. В случае медицинской диагностики вероятность описывает **степень веры**, причем 1 означает абсолютную уверенность в том, что у пациента грипп, а 0 – абсолютную уверенность в том, что у пациента нет гриппа. Первый вид вероятности, связанный с частотой возникновения событий, называется **частотной вероятностью**, а второй, связанный с качественным уровнем уверенности, – **байесовской вероятностью**.

Если перечислить несколько свойств, которыми в согласии со здравым смыслом должны обладать рассуждения о недостоверности, то окажется, что единственный способ удовлетворить эти свойства состоит в том, чтобы рассматривать поведение байесовских вероятностей в точности так же, как частотных. Например, при вычислении вероятности получения определенной комбинации в покере используются те же формулы, что при вычислении вероятности заболевания при наличии определенных симптомов. Подробнее о том, почему из небольшого набора основанных на здравом смысле предположений вытекают одни и те же аксиомы для обоих видов вероятности, см. Ramsey (1926).

Вероятность можно рассматривать как обобщение логики на рассуждения в условиях неопределенности. Логика дает нам набор формальных правил, позволяющих определить, истинно некоторое высказывание или ложно, в зависимости от предположения об истинности или ложности других высказываний. Теория вероятностей предлагает набор формальных правил для определения правдоподобия высказывания при условии правдоподобия других высказываний.

3.2. Случайные величины

Случайной величиной называется величина, случайно принимающая различные значения. Обычно сама случайная величина обозначается строчной буквой прямым шрифтом, а ее значения – строчными курсивными буквами. Например, x_1 и x_2 – возможные значения случайной величины x. Векторные случайные величины обозначаются **x**, а их значения – $\boldsymbol{x}$. Сама по себе случайная величина – это просто описание возможных состояний, ее можно использовать вместе с распределением вероятности, показывающим, насколько вероятно каждое состояние.

Случайные величины бывают дискретными и непрерывными. Дискретная случайная величина может иметь конечное или счетное число состояний. Отметим, что состояния – необязательно целые числа, допускаются также просто именованные состояния, с которыми не ассоциировано числовое значение. С непрерывной случайной величиной ассоциированы вещественные значения.

3.3. Распределения вероятности

Распределение вероятности описывает, с какой вероятностью случайная величина или множество случайных величин принимает каждое возможное значение. Способ задания распределения вероятности зависит от того, является случайная величина непрерывной или дискретной.

3.3.1. Дискретные случайные величины и функции вероятности

Распределение вероятности дискретной величины может быть описано с помощью **функции вероятности**. Обычно функции вероятности обозначаются прописной буквой *P*. Часто со случайными величинами ассоциируются разные функции вероятности, и читатель должен понимать, о какой функции идет речь, ориентируясь на природу случайной величины, а не на имя функции; $P(\mathrm{x})$ обычно не то же самое, что $P(\mathrm{y})$.

Функция вероятности отображает состояние случайной величины на вероятность того, что случайная величина находится в этом состоянии. Вероятность, что $\mathrm{x} = x$, обозначается $P(x)$, причем вероятность 1 означает, что событие $\mathrm{x} = x$ достоверно, а вероятность 0 – что оно невозможно. Иногда, чтобы недвусмысленно указать, какая функция вероятности используется, мы будем записывать имя случайной величины явно: $P(\mathrm{x} = x)$. В некоторых случаях мы сначала определяем случайную величину, а затем используем знак ~ для обозначения ее распределения: $\mathrm{x} \sim P(x)$.

Функции вероятности могут применяться сразу к нескольким величинам. Такая функция вероятности многих переменных называется **совместным распределением вероятности**. Запись $P(\mathrm{x} = x, \mathrm{y} = y)$ означает, что $\mathrm{x} = x$ и $\mathrm{y} = y$ одновременно. Для краткости мы иногда будем писать просто $P(x, y)$.

Чтобы функция *P* случайной величины могла рассматриваться как функция вероятности, она должна удовлетворять следующим условиям.

- Областью определения *P* является множество всех возможных состояний x.
- $\forall x \in \mathrm{x}\ 0 \leq P(x) \leq 1$. Невозможное событие имеет вероятность 0, и никакое состояние не может быть менее вероятно. Аналогично событие, которое произойдет наверняка, имеет вероятность 1, и никакое состояние не может быть более вероятно.
- $\sum_{x \in \mathrm{x}} P(x) = 1$. Это равенство называется свойством **нормировки**. Не будь его, можно было бы получить значение, большее 1, при вычислении вероятности одного из многих событий.

Рассмотрим, к примеру, одиночную дискретную случайную величины x с *k* состояниями. Мы можем задать равномерное распределение x – когда все состояния равновероятны, – определив функцию вероятности следующим образом:

$$P(\mathrm{x} = x_i) = \frac{1}{k} \tag{3.1}$$

для всех *i*. Это определение удовлетворяет всем требованиям к функции вероятности. Значение $1/k$ положительно, потому что *k* – положительное число. Кроме того,

$$\sum_i P(\mathrm{x} = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1, \tag{3.2}$$

так что распределение правильно нормировано.

3.3.2. Непрерывные случайные величины и функции плотности вероятности

Распределение непрерывных случайных величин описывается **функцией плотности вероятности** (ФПВ), а не функцией вероятности. Такая функция *p* должна обладать следующими свойствами.

- Область определения p – множество всех возможных состояний x.
- $\forall x \in \mathrm{x}, p(x) \geq 0$; отметим, что мы не требуем выполнения условия $p(x) \leq 1$.
- $\int p(x)dx = 1$.

Функция плотности вероятности $p(x)$ определяет не вероятность конкретного состояния, а вероятность попадания в бесконечно малую окрестность размера δx, которая равна $p(x)\delta x$.

Для нахождения вероятности множества точек следует проинтегрировать функцию плотности. Точнее, вероятность, что x принадлежит множеству $\mathbb{S}$, равна интегралу $p(x)$ по этому множеству. В одномерном случае вероятность принадлежности x отрезку $[a, b]$ равна $\int_{[a,b]} p(x)dx$.

В качестве примера рассмотрим равномерное распределение непрерывной случайной величины на отрезке вещественных чисел. Для этого нужно определить функции $u(x; a, b)$, где a и b – концы отрезка и $b > a$. Нотация «;» означает «параметризовано», т. е. x является аргументом функции, а a и b – ее параметрами. Чтобы гарантировать, что вся масса вероятности находится внутри отрезка, положим $u(x; a, b) = 0$ для всех $x \notin [a, b]$, а для точек внутри $[a, b]$ определим $u(x; a, b) = 1/(b - a)$. Легко видеть, что так определенная функция всюду неотрицательна. Кроме того, ее интеграл равен 1. Мы часто пишем $\mathrm{x} \sim U(a, b)$, желая сказать, что x равномерно распределена на $[a, b]$.

3.4. Маргинальное распределение вероятности

Иногда известно распределение вероятности множества величин, а мы хотим узнать распределение вероятности подмножества этих величин. Оно называется **маргинальным распределением вероятности**.

Предположим, к примеру, что есть две дискретные случайные величины x и y и известна функция $P(x, y)$. Для нахождения $P(x)$ можно воспользоваться **правилом сложения**:

$$\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_y P(\mathrm{x} = x, \mathrm{y} = y). \tag{3.3}$$

Название «маргинальное распределение» напоминает о процессе вычисления маргинальных вероятностей на полях[1] листа бумаги. Если записать значения $P(\mathrm{x}, \mathrm{y})$ в ячейках таблицы, строки которой соответствуют значениям x, а столбцы – значениям y, то будет естественно просуммировать по каждой строке таблицы и записать сумму $P(x)$ на полях справа от строки.

Для непрерывных величин суммирование заменяется интегрированием:

$$p(x) = \int p(x, y)dy. \tag{3.4}$$

3.5. Условная вероятность

Часто нас интересует вероятность некоторого события, при условии что произошло какое-то другое событие. Это называется **условной вероятностью**. Условная вероятность того, что $\mathrm{y} = y$ при условии, что $\mathrm{x} = x$, записывается в виде $P(\mathrm{y} = y \mid \mathrm{x} = x)$ и вычисляется по формуле:

---

[1] Английское margin – поле. – *Прим. перев.*

$$P(\mathrm{y} = y \mid \mathrm{x} = x) = \frac{P(\mathrm{y} = y, \mathrm{x} = x)}{P(\mathrm{x} = x)}. \tag{3.5}$$

Условная вероятность определена только тогда, когда $P(\mathrm{x} = x) > 0$. Невозможно вычислить условную вероятность при условии события, которое не может произойти.

Важно не путать условную вероятность с вычислением того, что могло бы произойти, если бы было предпринято некоторое действие. Условная вероятность того, что человек родом из Германии, если он говорит по-немецки, довольно высока, но если случайно выбранного человека научить говорить по-немецки, то страна его рождения не изменится. Вычисление последствий действия называется **запросом о вмешательстве** и составляет предмет **причинного моделирования**, которое в этой книге не рассматривается.

3.6. Цепное правило

Любое совместное распределение вероятности нескольких случайных величин можно разложить в произведение условных вероятностей одной величины:

$$P(\mathrm{x}^{(1)}, ..., \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)}) \prod_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)}, ..., \mathrm{x}^{(i-1)}). \tag{3.6}$$

Эта формула называется **цепным правилом**, или **правилом умножения** вероятностей. Она сразу следует из определения условной вероятности (3.5). Применив его дважды, получим:

$$P(\mathrm{a}, \mathrm{b}, \mathrm{c}) = P(\mathrm{a} \mid \mathrm{b}, \mathrm{c}) P(\mathrm{b}, \mathrm{c})$$
$$P(\mathrm{b}, \mathrm{c}) = P(\mathrm{b} \mid \mathrm{c}) P(\mathrm{c})$$
$$P(\mathrm{a}, \mathrm{b}, \mathrm{c}) = P(\mathrm{a} \mid \mathrm{b}, \mathrm{c}) P(\mathrm{b} \mid \mathrm{c}) P(\mathrm{c})$$

3.7. Независимость и условная независимость

Две случайные величины x и y называются **независимыми**, если их совместное распределение вероятности можно представить в виде произведения двух сомножителей, один из которых содержит только x, а другой – только y:

$$\forall x \in \mathrm{x}, y \in \mathrm{y}, p(\mathrm{x} = x, \mathrm{y} = y) = p(\mathrm{x} = x) p(\mathrm{y} = y). \tag{3.7}$$

Две случайные величины x и y называются **условно независимыми** при условии случайной величины z, если условное распределение вероятности x и y можно следующим образом представить в виде произведения для любого значения z:

$$\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z}, p(\mathrm{x} = x, \mathrm{y} = y \mid \mathrm{z} = z) = p(\mathrm{x} = x \mid \mathrm{z} = z) p(\mathrm{y} = y \mid \mathrm{z} = z). \tag{3.8}$$

Для обозначения независимости и условной независимости будем применять компактную нотацию: $\mathrm{x} \perp \mathrm{y}$ означает, что x и y независимы, а $\mathrm{x} \perp \mathrm{y} \mid \mathrm{z}$ – что x и y условно независимы при условии z.

3.8. Математическое ожидание, дисперсия и ковариация

Математическим ожиданием, или **ожидаемым значением**, функции $f(x)$ относительно распределения вероятности $P(\mathrm{x})$ называется среднее значение f, когда x вы-

бирается из P. Для дискретных величин математическое ожидание вычисляется суммированием:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x) f(x), \tag{3.9}$$

а для непрерывных – интегрированием:

$$\mathbb{E}_{x\sim P}[f(x)] = \int p(x) f(x) dx. \tag{3.10}$$

Если характер распределения ясен из контекста, то можно просто написать имя случайной величины, по которой вычисляется математическое ожидание: $\mathbb{E}_x[f(x)]$. А если и относительно случайной величины не возникает сомнений, то можно вообще опустить подстрочный индекс: $\mathbb{E}[f(x)]$. По умолчанию предполагается, что $\mathbb{E}[\cdot]$ производит усреднение по всем случайным величинам в квадратных скобках. Если не возникает двусмысленности, то квадратные скобки можно опускать.

Операция математического ожидания линейна, т. е.

$$\mathbb{E}_x[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}_x[f(x)] + \beta \mathbb{E}_x[g(x)], \tag{3.11}$$

где α и β не зависят от x.

Дисперсия измеряет разброс значений функции случайной величины x с заданным распределением вероятности:

$$\mathrm{Var}(f(x)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]. \tag{3.12}$$

Если дисперсия мала, то значения $f(x)$ сгруппированы в окрестности ожидаемого значения. Квадратный корень из дисперсии называется **стандартным отклонением**.

Ковариация дает представление о силе линейной связи между двумя функциями случайных величин, а также о масштабе этих величин:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]. \tag{3.13}$$

Если абсолютная величина ковариации высока, то значения различаются очень сильно и одновременно далеки от своих средних. Если ковариация положительна, обе величины принимают большие значения одновременно, а если отрицательна, то в тот момент, когда одна величина принимает большие значения, другая принимает малые значения (и наоборот). Другие метрики, например **корреляция**, нормируют вклад каждой величины, чтобы измерить только связь между самими величинами, не включая в рассмотрение их масштабы.

Понятия ковариации и зависимости взаимосвязаны, но различны. Связаны они потому, что две независимые случайные величины имеют нулевую ковариацию, а две величины с ненулевой ковариацией зависимы. Однако независимость отличается от ковариации. Чтобы две величины имели нулевую ковариацию, между ними не должно быть линейной зависимости. Независимость же – более строгое требование, чем нулевая ковариация, поскольку она исключает также наличие нелинейных связей. Может случиться так, что две случайные величины зависимы, но их ковариация равна нулю. Предположим, к примеру, что мы сначала выбрали вещественное число x из равномерного распределения на отрезке $[-1, 1]$. Затем выбираем вторую случайную величину s. С вероятностью $^1/_2$ значение s равно 1, иначе -1. Далее порождаем случайную величину $y = sx$. Очевидно, что x и y не являются независимыми, поскольку x однозначно определяет абсолютную величину y. Однако же $\mathrm{Cov}(x, y) = 0$.

Ковариационной матрицей случайного вектора $\boldsymbol{x} \in \mathbb{R}^n$ называется матрица размера $n \times n$ – такая, что

$$\mathrm{Cov}(\mathbf{x})_{i,j} = \mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j). \tag{3.14}$$

Диагональные элементы ковариационной матрицы равны дисперсии:

$$\mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j) = \mathrm{Var}(\mathrm{x}_i). \tag{3.15}$$

3.9. Часто встречающиеся распределения вероятности

В контексте машинного обучения полезно несколько простых распределений вероятности.

3.9.1. Распределение Бернулли

Распределение Бернулли – это распределение одной случайной величины, принимающей всего два значения. У него имеется единственный параметр $\phi \in [0, 1]$, определяющий, с какой вероятностью случайная величина равна 1. Это распределение обладает следующими свойствами:

$$P(\mathrm{x} = 1) = \phi \tag{3.16}$$

$$P(\mathrm{x} = 0) = 1 - \phi \tag{3.17}$$

$$\mathrm{P}(\mathrm{x} = x) = \phi^x(1 - \phi)^{1-x} \tag{3.18}$$

$$\mathbb{E}_x[\mathrm{x}] = \phi \tag{3.19}$$

$$\mathrm{Var}_x(\mathrm{x}) = \phi(1 - \phi) \tag{3.20}$$

3.9.2. Категориальное распределение

Категориальным распределением (или распределением «multinoulli») называется распределение одной дискретной случайной величины, принимающей конечное число значений k[1]. Категориальное распределение параметризовано вектором $\boldsymbol{p} \in [0, 1]^{k-1}$, где p_i – вероятность i-го состояния. Вероятность последнего, k-го, состояния равна $1 - \mathbf{1}^\top \boldsymbol{p}$. При этом необходимо наложить ограничение $\mathbf{1}^\top \boldsymbol{p} \leq 1$. Категориальные распределения часто используются для описания распределения категорий объектов, поэтому мы обычно не будем предполагать, что состоянию 1 соответствует числовое значение 1 и т. д. По этой причине нам, как правило, не нужно вычислять математическое ожидание или дисперсию случайных величин с категориальным распределением.

Двух распределений – Бернулли и категориального – достаточно для описания любого распределения в соответствующей области определения. Это так не потому, что они такие общие, а потому, что область определения проста; моделируются

---

[1] Термин «multinoulli» недавно предложил Густаво Ласердо и популяризовал Мэрфи (Murphy 2012). Это частный случай **мультиномиального распределения**, т. е. распределения векторов $\{0, ..., n\}^k$, описывающего, сколько раз встретится каждая из k категорий при выборке объема n из категориального распределения. Во многих источниках словом «мультиномиальное» обозначают категориальное распределение, не уточняя, что речь идет только о случае $n = 1$.

дискретные величины, для которых можно перечислить все возможные состояния. В случае непрерывных случайных величин множество состояний несчетно, поэтому любое распределение, описываемое небольшим количеством параметров, по необходимости налагает строгие ограничения.

3.9.3. Нормальное распределение

Самым распространенным распределением вещественных чисел является **нормальное**, или **гауссово**, **распределение**:

$$\mathcal{N}(x;\mu,\sigma^2)=\sqrt{\frac{1}{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.21}$$

График функции плотности нормального распределения показан на рис. 3.1.

У нормального распределения два параметра: $\mu \in \mathbb{R}$ и $\sigma \in (0, \infty)$. Параметр μ определяет абсциссу центрального пика и одновременно является средним значением распределения: $\mathbb{E}[x] = \mu$. Стандартное отклонение этого распределения равно σ, а дисперсия – σ^2.

В выражение для функции плотности вероятности входит обратный квадрат σ. Если требуется часто вычислять ФПВ с разными параметрами, то эффективнее параметризовать распределение параметром $\beta \in (0, \infty)$, который задает **точность**, или обратную дисперсию распределения:

$$\mathcal{N}(x;\mu,\beta^{-1})=\sqrt{\frac{\beta}{2\pi}}\exp\left(-\frac{1}{2}\beta(x-\mu)^2\right). \tag{3.22}$$

Нормальные распределения являются разумным выбором для многих приложений. В отсутствие априорных знаний о форме распределения вещественных чисел нормальное распределение выбирается по умолчанию в силу двух основных причин:

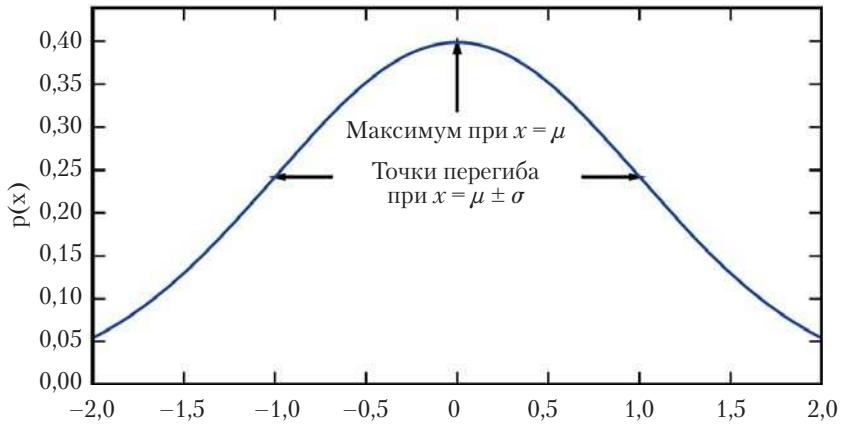

Рис. 3.1 ❖ Нормальное распределение $\mathcal{N}(x; \mu, \sigma^2)$ имеет классическую колоколообразную форму, при этом абсцисса центрального пика задается параметром μ, а ширина пика определяется параметром $\mathcal{N}$. На этом рисунке показано стандартное нормальное распределение с $\mu = 0$, $\sigma = 1$

Во-первых, многие моделируемые распределения действительно близки к нормальному. Согласно **центральной предельной теореме**, сумма многих независимых случайных величин аппроксимируется нормальным распределением. На практике это означает, что многие сложные системы успешно моделируются как нормально распределенный шум, даже если систему можно разложить на части с более структурированным поведением.

Во-вторых, из всех возможных распределений вероятности вещественных чисел с одной и той же дисперсией нормальное распределение обладает максимальной неопределенностью. То есть можно считать, что нормальное распределение привносит в модель минимум априорных знаний. Для развития и обоснования этой идеи необходим дополнительный математический аппарат, поэтому мы отложим это до раздела 19.4.2. Нормальное распределение обобщается на пространство $\mathbb{R}^n$ и тогда называется **многомерным нормальным распределением**. Оно параметризуется положительно определенной симметричной матрицей $\boldsymbol{\Sigma}$:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sqrt{\frac{1}{(2\pi)^n \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right). \tag{3.23}$$

Параметр $\boldsymbol{\mu}$ по-прежнему задает среднее значение распределения, хотя теперь это вектор. Параметр $\boldsymbol{\Sigma}$ определяет ковариационную матрицу распределения. Как и в одномерном случае, если требуется вычислять ФПВ многократно с разными параметрами, то ковариационная матрица – не лучший способ параметризации, поскольку для вычисления ФПВ ее приходится обращать. Лучше использовать **матрицу точности** β:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \sqrt{\frac{\det(\boldsymbol{\beta})}{(2\pi)^n}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\beta}(\boldsymbol{x}-\boldsymbol{\mu})\right). \tag{3.24}$$

Часто ковариационная матрица является диагональной. Еще более простой случай – **изотропное нормальное распределение**, когда ковариационная матрица не только диагональная, но и все элементы на диагонали одинаковы.

3.9.4. Экспоненциальное распределение и распределение Лапласа

В контексте глубокого обучения нам часто необходимо распределение вероятности с острым пиком в точке $x = 0$. Для этого подходит экспоненциальное распределение:

$$p(x, \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x). \tag{3.25}$$

В определении экспоненциального распределения используется индикаторная функция $\mathbf{1}_{x \geq 0}$, равная нулю для всех отрицательных значений x.

С экспоненциальным тесно связано распределение Лапласа, позволяющее расположить пик массы вероятности в произвольной точке μ:

$$\text{Laplace}(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x-\mu|}{\gamma}\right). \tag{3.26}$$

3.9.5. Распределение Дирака и эмпирическое распределение

В некоторых случаях мы хотим иметь распределение, в котором вся масса вероятности сосредоточена в одной точке. Для этого можно использовать дельта-функцию Дирака $\delta(x)$:

$$p(x) = \delta(x - \mu). \tag{3.27}$$

Дельта-функция Дирака определена таким образом, что равна нулю всюду, кроме точки 0, но тем не менее ее интеграл равен 1. Это не обычная функция, которая сопоставляет каждому значению x некоторое вещественное значение, а иной математический объект – **обобщенная функция**, определяемая в терминах свойств интегрируемости. Дельта-функцию Дирака можно представлять себе как предел последовательности функций, которые оставляют все меньше и меньше массы во всех точках, кроме нуля.

Определив $p(x)$ как сдвиг δ на величину $-\mu$, мы получили бесконечно узкий и бесконечно высокий пик массы вероятности в точке $x = \mu$.

Дельта-распределение Дирака часто применяется в качестве компоненты **эмпирического распределения**:

$$\hat{p}(\boldsymbol{x}) = \frac{1}{m}\sum_{i=1}^{m}\delta(\boldsymbol{x} - \boldsymbol{x}^{(i)}), \tag{3.28}$$

которое сопоставляет массу вероятности $1/m$ каждой из точек $x^{(1)}$, ..., $x^{(m)}$, образуя заданный набор примеров. Дельта-распределение Дирака нужно только для определения эмпирического распределения непрерывных величин. Для дискретных величин все проще: эмпирическое распределение можно концептуально представить как категориальное распределение, в котором вероятность каждого возможного входного значения просто равна эмпирической частоте этого значения в обучающем наборе.

Эмпирическое распределение, образованное набором обучающих примеров, можно рассматривать как распределение, из которого производится выборка при обучении модели на этом наборе. Еще один важный взгляд на эмпирическое распределение заключается в том, что это плотность вероятности, доставляющая максимум правдоподобию обучающих данных (см. раздел 5.5).

3.9.6. Смеси распределений

Часто новые распределения вероятности определяются путем комбинирования более простых. Один из самых употребительных способов комбинирования – построение **смеси распределений**, состоящей из нескольких компонент. При каждом испытании определяется, из какого распределения будет производиться выборка, причем для выборки компоненты используется категориальное распределение:

$$P(\mathrm{x}) = \sum_{i} P(\mathrm{c} = i)P(\mathrm{x} \mid \mathrm{c} = i), \tag{3.29}$$

где $P(c)$ – категориальное распределение компонент.

Один пример смеси распределений мы уже видели: эмпирическое распределение вещественных случайных величин – это смесь распределений Дирака, по одному для каждого обучающего примера.

Модель смеси – простая стратегия комбинирования распределений вероятности для получения более сложного распределения. В главе 16 мы рассмотрим искусство построения сложных распределений из простых более подробно.

Модель смеси позволяет предварительно познакомиться с концепцией, которая в дальнейшем будет играть важнейшую роль – **скрытой**, или **латентной**, переменной. Латентной называется случайная величина, которую мы не наблюдаем непосредственно. Примером может служить величина c, определяющая компоненту смеси распределений. Латентные переменные могут быть связаны с x совместным распределением, в таком случае $P(\mathrm{x}, \mathrm{c}) = P(\mathrm{x} \mid \mathrm{c})P(\mathrm{c})$. Распределение $P(\mathrm{c})$ латентной переменной и распределение $P(\mathrm{x} \mid \mathrm{c})$, связывающее латентные переменные с видимыми, определяют форму распределения $P(\mathrm{x})$, пусть даже имеется возможность описать $P(\mathrm{x})$, не прибегая к латентной переменной. Латентные переменные будут обсуждаться в разделе 16.5.

Очень полезна и широко употребляется модель гауссовой смеси, в которой компоненты $p(\mathbf{x} \mid \mathrm{c} = i)$ – нормальные распределения. Каждая компонента независимо параметризуется средним значением $\boldsymbol{\mu}^{(i)}$ и ковариацией $\boldsymbol{\Sigma}^{(i)}$. На некоторые смеси могут налагаться дополнительные ограничения. Например, у разных компонент могут быть одинаковые ковариации, т. е. $\boldsymbol{\Sigma}^{(i)} = \boldsymbol{\Sigma}, \forall i$. Как и в случае одного нормального распределения, можно ограничиться только такими смесями, для которых ковариационная матрица каждой компоненты диагональная или изотропная.

Помимо средних и ковариаций, гауссова смесь параметризуется **априорными вероятностями** $\alpha_i = P(c = i)$ каждой компоненты i. Слово «априорная» означает, что речь идет о гипотезе относительно значения с до наблюдения $\mathbf{x}$. Напротив, $P(\mathrm{c} \mid \boldsymbol{x})$ называется **апостериорной вероятностью**, потому что вычисляется после наблюдения $\mathbf{x}$. Модель гауссовой смеси является универсальным приближением для плотностей в том смысле, что любую гладкую функцию плотности можно аппроксимировать с любой наперед заданной точностью смесью нормальных распределений с достаточно большим числом компонент.

На рис. 3.2 показаны выборки из модели гауссовой смеси.

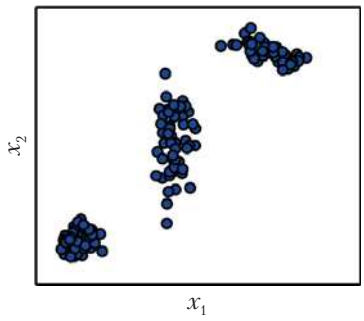

Рис. 3.2 ❖ Выборки из модели гауссовой смеси. Показаны три компоненты. Если смотреть слева направо, то первая компонента имеет изотропную ковариационную матрицу, т. е. дисперсия во всех направлениях одинакова. Вторая компонента имеет диагональную ковариационную матрицу, т. е. дисперсию можно задавать независимо вдоль каждой оси. В этом примере дисперсия вдоль оси x_2 больше, чем вдоль оси x_1. Третья компонента имеет ковариационную матрицу полного ранга, позволяющую задавать дисперсии вдоль произвольного базиса

3.10. Полезные свойства употребительных функций

При работе с распределениями вероятности, особенно в контексте глубокого обучения, некоторые функции встречаются очень часто.

Одна из них – **логистическая сигмоида**:

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{3.30}$$

Логистическая сигмоида обычно используется для порождения параметра ϕ распределения Бернулли, поскольку она принимает значения из интервала (0, 1), принадлежащего области допустимых значений параметра ϕ. На рис. 3.3 показан график сигмоидной функции. Как видно, если абсолютное значение аргумента велико, то функция **выходит на плато**, т. е. становится очень плоской и малочувствительной к небольшим изменениям аргумента.

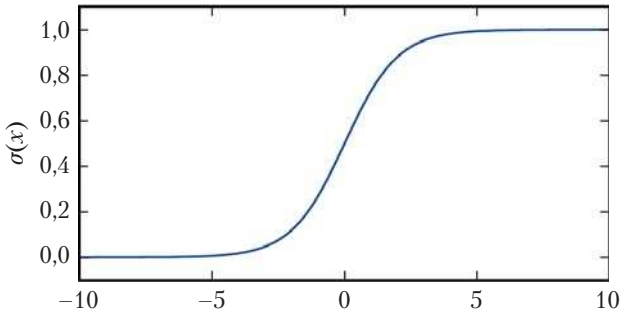

Рис. 3.3 ❖ Логистическая сигмоида

Часто встречается также **функция softplus** (Dugas et al., 2001):

$$\zeta(x) = \log(1 + \exp(x)). \tag{3.31}$$

Эта функция может быть полезна для порождения параметра β или α нормального распределения, поскольку принимает значения из интервала $(0, \infty)$. Она также нередко возникает при работе с выражениями, содержащими сигмоиды. Название softplus объясняется тем, что это сглаженный (смягченный – «softened») вариант функции

$$x^+ = \max(0, x). \tag{3.32}$$

График функции softplus показан на рис. 3.4.

Приведенные ниже свойства настолько полезны, что имеет смысл запомнить их:

$$\sigma(x) = \frac{\exp(x)}{\exp(x)+\exp(0)} \tag{3.33}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x)) \tag{3.34}$$

$$1 - \sigma(x) = \sigma(-x) \tag{3.35}$$

$$\log \sigma(x) = -\zeta(-x) \tag{3.36}$$

$$\frac{d}{dx}\zeta(x) = \sigma(x) \tag{3.37}$$

$$\forall x \in (0,1),\ \sigma^{-1}(x) = \log\left(\frac{x}{1-x}\right) \tag{3.38}$$

$$\forall x > 0,\ \zeta^{-1}(x) = \log(\exp(x) - 1) \tag{3.39}$$

$$\zeta(x) = \int_{-\infty}^{x} \sigma(y)dy \tag{3.40}$$

$$\zeta(x) - \zeta(-x) = x \tag{3.41}$$

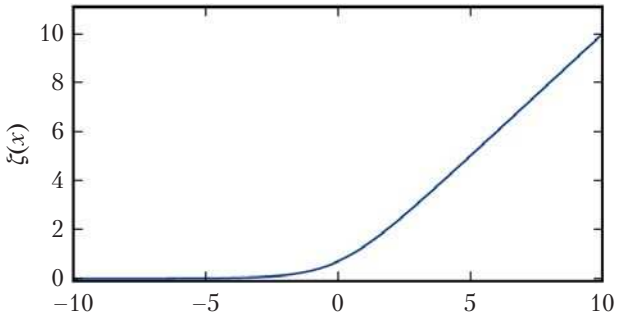

Рис. 3.4 ❖ Функция softplus

В статистике функция $\sigma^{-1}(x)$ называется **logit**, но в машинном обучении этот термин используется редко.

Формула (3.41) дает еще одно обоснование названия «softplus». Назначение функции softplus – служить сглаженным вариантом функции положительной части $x^+ = \max\{0, x\}$, дополнением к которой является функция отрицательной части $x^- = \max\{0, -x\}$. Для сглаживания отрицательной части можно взять функцию $\zeta(-x)$. Величину x можно восстановить по положительной и отрицательной частям благодаря тождеству $x^+ - x^- = x$, а в силу тождества (3.41) x можно восстановить также по $\zeta(x)$ и $\zeta(-x)$.

3.11. Правило Байеса

Часто возникает ситуация, когда известна вероятность $P(\mathrm{y} \mid \mathrm{x})$, а требуется узнать $P(\mathrm{x} \mid \mathrm{y})$. К счастью, если мы знаем также $P(\mathrm{x})$, то можем вычислить искомую вероятность по **правилу Байеса**:

$$P(\mathrm{x} \mid \mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y} \mid \mathrm{x})}{P(\mathrm{y})}. \tag{3.42}$$

Отметим, что в эту формулу входит $P(\mathrm{y})$, но обычно можно вычислить $P(\mathrm{y}) = \Sigma_x P(\mathrm{y} \mid x)P(x)$, так что без знания $P(\mathrm{y})$ можно обойтись.

Правило Байеса непосредственно следует из определения условной вероятности, но знать название этой формулы полезно, потому что оно часто встречается в различных публикациях. Формула названа в честь преподобного Томаса Байеса, который первым открыл ее частный случай. В общем виде, представленном выше, ее независимо открыл Пьер-Симон Лаплас.

3.12. Технические детали непрерывных величин

Для формального изложения непрерывных случайных величин и функций плотности вероятности необходимо знакомство с разделом математики, который называется **теория меры**. Эта теория выходит за рамки книги, но мы можем бегло осветить вопросы, которые она решает.

В разделе 3.3.2 мы видели, что вероятность нахождения непрерывной векторной случайной величины $\mathbf{x}$ в некотором множестве $\mathbb{S}$ равна интегралу функции $p(\boldsymbol{x})$, взятому по этому множеству. Но при определенном выборе $\mathbb{S}$ возникают парадоксы. Например, можно построить два множества $\mathbb{S}_1$ и $\mathbb{S}_2$, так что $p(\boldsymbol{x} \in \mathbb{S}_1) + p(\boldsymbol{x} \in \mathbb{S}_2) > 1$, но $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$. При построении таких множеств используются инфинитезимальные свойства вещественных чисел, например строятся фрактальные множества или множества, получаемые преобразованием множества рациональных чисел[1]. Один из основных вкладов теории меры – характеристика множества множеств, для которых можно вычислить вероятность, не сталкиваясь с парадоксами. В этой книге интегрирование производится только по относительно простым множествам, так что этот аспект теории меры не имеет большого значения.

Для наших целей теория меры более полезна с точки зрения теорем, применимых к большинству точек $\mathbb{R}^n$, за исключением некоторых граничных случаев. Теория меры позволяет строго описать, что такое пренебрежимо малое множество точек. Говорят, что такое множество имеет **меру нуль**. Мы здесь не станем формально определять это понятие. Достаточно интуитивного представления о том, что множество меры нуль не занимает никакого объема в рассматриваемом пространстве. Например, в $\mathbb{R}^2$ прямая имеет меру нуль, а залитый многоугольник – положительную меру. Точно так же отдельно взятая точка имеет меру нуль. Объединение счетного множества множеств меры нуль само имеет меру нуль (так что множество всех рациональных чисел имеет меру нуль).

Еще один полезный термин из теории меры – **почти всюду**. Говорят, что свойство выполняется почти всюду, если оно выполняется для всех точек пространства, за исключением множества меры нуль. Поскольку исключения занимают пренебрежимо малую часть пространства, в большинстве приложений их можно без опаски игнорировать. Некоторые важные результаты теории вероятностей справедливы для всех дискретных случайных величин, но лишь «почти всюду» для непрерывных.

Еще одна техническая деталь, относящаяся к непрерывным величинам, – работа с непрерывными случайными величинами, являющимися детерминированными функциями других случайных величин. Допустим, имеются две случайные величины $\mathbf{x}$ и $\mathbf{y}$ – такие, что $\boldsymbol{y} = g(\boldsymbol{x})$, где g – обратимое, непрерывное, дифференцируемое преобразование. Можно было бы ожидать, что $p_y(\boldsymbol{y}) = p_x(g^{-1}(\boldsymbol{y}))$. На самом деле это не так.

В качестве простого примера предположим, что имеются скалярные случайные величины x и y. Пусть $y = x / 2$ и $x \sim U(0, 1)$. Если бы имело место тождество $p_y(y) = p_x(2y)$, то p_y было бы равно 0 всюду, кроме отрезка $[0, 1/2]$, и 1 на этом отрезке. Тогда

$$\int p_y(y)dy = \frac{1}{2}, \tag{3.43}$$

что противоречит определению распределения вероятности. Это типичная ошибка. Проблема в том, что при таком подходе не учитывается искажение пространства, вно-

[1] Забавный пример таких множеств дает теорема Банаха–Тарского.

симое функцией g. Напомним, что вероятность попадания $\boldsymbol{x}$ в бесконечно малую область объема $\delta\boldsymbol{x}$ равна $p(\boldsymbol{x})\delta\boldsymbol{x}$. Поскольку g может расширять или сжимать пространство, бесконечно малая окрестность $\boldsymbol{x}$ в пространстве $\boldsymbol{x}$ может иметь другой объем в пространстве $\boldsymbol{y}$.

Чтобы понять, как справиться с этой трудностью, вернемся к скалярному случаю. Нам требуется сохранить свойство

$$|p_y(g(x))dy| = |p_x(x)dx|. \tag{3.44}$$

Решая это уравнение, получаем

$$p_y(y) = p_x(g^{-1}(y))\left|\frac{\partial x}{\partial y}\right|, \tag{3.45}$$

или эквивалентно

$$p_x(x) = p_y(g(x))\left|\frac{\partial g(x)}{\partial x}\right|. \tag{3.46}$$

В пространстве более высокой размерности обобщением производной является определитель **матрицы Якоби**: $J_{i,j} = \partial x_i / \partial y_j$. Следовательно, для вещественных векторов $\boldsymbol{x}$ и $\boldsymbol{y}$

$$p_x(\boldsymbol{x}) = p_y(g(\boldsymbol{x}))\left|\det\left(\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}}\right)\right|. \tag{3.47}$$

3.13. Теория информации

Теория информации – это раздел прикладной математики, в котором изучается количественное содержание информации в сигнале. Первоначально она была создана для изучения передачи сообщений, состоящих из символов дискретного алфавита, по зашумленному каналу, например в случае радиопередачи. В этом контексте теория информации подсказывает, как построить оптимальный код и вычислить ожидаемую длину сообщений, выбираемых из конкретного распределения вероятности при различных схемах кодирования. В контексте машинного обучения теорию информации можно применить также к непрерывным величинам, когда некоторые интерпретации длины сообщения бессмысленны. Теория информации является основополагающей для многих разделов электротехники и информатики. В этой книге мы будем пользоваться лишь немногими ключевыми идеями теории информации, чтобы охарактеризовать распределения вероятности или количественно измерить сходство между двумя распределениями. Дополнительные сведения см. в книгах Cover and Thomas (2006) или MacKay (2003).

В основе теории информации лежит интуитивное соображение, согласно которому в наблюдении маловероятного события содержится больше информации, чем в наблюдении вероятного события. Сообщение «сегодня утром взошло солнце» настолько неинформативно, что его и передавать не нужно, а вот сообщение «сегодня утром было солнечное затмение» очень информативно.

Мы хотели бы количественно измерить информацию, чтобы формализовать эту идею.

- Вероятные события должны иметь низкое информационное содержание, а если событие наступает гарантированно, то информационное содержание отсутствует полностью.
- Чем меньше вероятность события, тем выше информационное содержание.
- Информация, содержащаяся в независимых событиях, складывается. Например, событие, заключающееся в выпадении орла два раза, должно содержать в два раза больше информации, чем событие однократного выпадения орла.

Чтобы удовлетворить все три свойства, определим **собственную информацию** события $\mathrm{x} = x$:

$$I(x) = -\log P(x). \tag{3.48}$$

В этой книге log всегда означает натуральный логарифм (по основанию e). Поэтому в нашем определении $I(x)$ измеряется в **натах**. Один нат равен количеству информации, содержащемуся в наблюдении события с вероятностью $1/e$. В других учебниках используются логарифмы по основанию 2, соответствующие единицы измерения называются **битами**, или **шеннонами**; информация, измеренная в битах, отличается от измеренной в натах только постоянным коэффициентом.

Если x – непрерывная случайная величина, то по аналогии используется точно такое же определение информации, но некоторые свойства, справедливые в дискретном случае, утрачиваются. Например, событие с единичной плотностью содержит нулевую информацию, хотя и не является достоверным.

Собственная информация относится только к одному исходу. Мы можем количественно выразить неопределенность всего распределения вероятности, воспользовавшись **энтропией Шеннона**

$$H(x) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log P(x)], \tag{3.49}$$

которая обозначается также $H(P)$. Иными словами, энтропия Шеннона распределения – это математическое ожидание количества информации в событии, выбираемом из этого распределения. Энтропия дает нижнюю границу числа бит (если логарифм берется по основанию 2, в противном случае единицы измерения иные), необходимое в среднем для кодирования символов, выбираемых из распределения P. Почти детерминированные распределения (когда исход испытания почти достоверен) имеют низкую энтропию; распределения, близкие к равномерному, – высокую. Энтропия иллюстрируется на рис. 3.5. Если x – непрерывная величина, то энтропия Шеннона называется **дифференциальной энтропией**.

Если $P(\mathrm{x})$ и $Q(\mathrm{x})$ – два распределения вероятности одной и той же случайной величины x, то измерить, насколько они различаются, позволяет **расхождение Кульбака–Лейблера (КЛ)**:

$$D_{\mathrm{KL}}(P \| Q) = \mathbb{E}_{\mathrm{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathrm{x}\sim P}[\log P(x) - \log Q(x)]. \tag{3.50}$$

Для дискретных величин это дополнительное количество информации (измеренное в битах, если используется логарифм по основанию 2, но в машинном обучении обычно используются натуральный логарифм и наты), необходимое для отправки сообщения, содержащего символы, выбранные из распределения P, если используется код, спроектированный для минимизации длины сообщений, выбираемых из распределения Q.

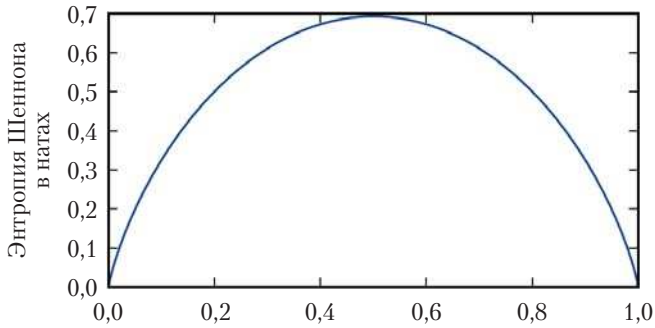

Рис. 3.5 ❖ Энтропия Шеннона бинарной случайной величины. На графике видно, что у распределений, близких к детерминированному, энтропия низкая, а у близких к равномерному – высокая. По горизонтальной оси отложена вероятность p того, что бинарная случайная величина близка к 1. Энтропия вычисляется по формуле $(p-1)\log(1-p) - p\log p$. Если p близка к 0, то распределение почти детерминировано, поскольку случайная величина почти всегда равна 0. Если p близка к 1, то распределение почти детерминировано, поскольку случайная величина почти всегда равна 1. При $p = 0{,}5$ энтропия максимальна, поскольку оба исхода распределены равномерно

Расхождение КЛ обладает многими полезными свойствами, самое главное из которых – неотрицательность.

Расхождение КЛ равно 0 тогда и только тогда, когда P и Q – одно и то же распределение в случае дискретных величин или когда эти распределения совпадают «почти всюду» – в случае непрерывных. Поскольку расхождение КЛ неотрицательно и измеряет различие между двумя распределениями, его часто концептуально воспринимают как средство измерения некоторого расстояния между распределениями. Но, строго говоря, эта мера – не расстояние, поскольку не является симметричной: $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$ для некоторых P и Q. Наличие асимметрии означает, что от выбора $D_{\mathrm{KL}}(P\|Q)$ или $D_{\mathrm{KL}}(Q\|P)$ многое зависит. Детали показаны на рис. 3.6.

С расхождением КЛ тесно связана **перекрестная энтропия** $H(P, Q) = H(P) + D_{\mathrm{KL}}(P\|Q)$, аналогичная расхождению КЛ, но без левого члена:

$$H(P, Q) = -\mathbb{E}_{x\sim P}\log Q(x). \tag{3.51}$$

Минимизация перекрестной энтропии относительно Q равносильна минимизации расхождения КЛ, потому что Q не входит в опущенный член.

При вычислении подобных величин часто встречаются выражения вида $0\log 0$. По принятому в теории информации соглашению, они трактуются как $\lim_{x\to 0} x\log x = 0$.

3.14. Структурные вероятностные модели

В алгоритмах машинного обучения часто встречаются распределения вероятности очень большого числа случайных величин. Нередко в этих распределениях наличествуют прямые взаимодействия между сравнительно небольшим количеством переменных. Использовать одну функцию для описания всего совместного распределения вероятности может оказаться очень неэффективно (и с вычислительной, и со статистической точки зрения).

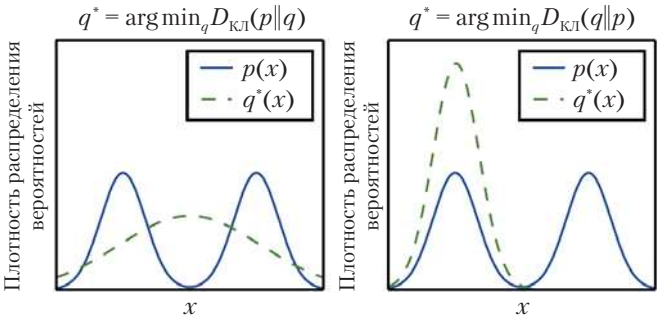

Рис. 3.6 ❖ Расхождение КЛ несимметрично. Предположим, что мы хотим аппроксимировать распределение $p(x)$ другим распределением $q(x)$. Можно выбирать, что минимизировать: $D_{KL}(p\|q)$ или $D_{KL}(q\|p)$. На рисунке показаны последствия выбора в случае, когда p – смесь двух нормальных распределений, а q – обычное нормальное распределение. Выбор направления расхождения зависит от задачи. Для одних приложений нужна аппроксимация, в которой вероятность высока там, где высока вероятность истинного распределения, а для других – чтобы была низкая вероятность там, где низка вероятность истинного распределения. (*Слева*) Результат минимизации $D_{KL}(p\|q)$. В этом случае q выбирается так, чтобы была высокая вероятность там, где высока вероятность p. Если p имеет несколько мод, то q стремится размазать моды, собрав заключенную в них массу вероятности. (*Справа*) Результат минимизации $D_{KL}(q\|p)$. В этом случае q выбирается так, чтобы была низкая вероятность там, где низка вероятность p. Если p имеет несколько достаточно далеко отстоящих мод, как на этом рисунке, то расхождение КЛ достигает минимума, когда выбирается одна мода, чтобы предотвратить размещение массы вероятности в областях низкой вероятности между модами p. На рисунке показан результат, когда q выбрано, так чтобы усилить левую моду. Такое же значение расхождения КЛ можно было бы получить, выбрав правую моду. Если моды не разделены достаточно выраженной областью малой вероятности, то и при таком выборе направления расхождения КЛ может произойти размазывание мод

Вместо этого мы можем разделить распределение вероятности на много перемножаемых факторов. Допустим, к примеру, что имеются три случайные величины: a, b и c. Предположим, что a влияет на b, b влияет на c, но a и c независимы при условии b. Распределение вероятности всех трех переменных можно представить в виде произведения распределений двух переменных:

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}) = p(\mathrm{a})p(\mathrm{b}|\mathrm{a})p(\mathrm{c}|\mathrm{b}). \tag{3.52}$$

Такая факторизация может существенно уменьшить число параметров, необходимых для описания распределения. Число параметров каждого фактора экспоненциально зависит от числа переменных в нем. Это значит, что стоимость представления распределения удастся значительно сократить, если мы сможем разложить его в произведение распределений с меньшим числом переменных.

Подобные факторизации можно описывать с помощью графов; под графом здесь понимается множество вершин, некоторые из которых соединены ребрами. Если факторизация распределения вероятности представлена в виде графа, то мы называем его **структурной вероятностной моделью**, или **графической моделью**.

Существуют два основных вида структурных вероятностных моделей: ориентированные и неориентированные. В обоих случаях используется граф $\mathcal{G}$, в котором каждая вершина соответствует случайной величине, а наличие ребра между двумя вершинами означает, что распределение вероятности способно представить прямые взаимодействия между соответствующими величинами.

В **ориентированных** моделях используются графы с ориентированными ребрами, они представляют разложение в произведение условных распределений вероятности, как в примере выше. Точнее, ориентированная модель содержит по одному фактору для каждой случайной величины x_i в распределении, и этот фактор состоит из условного распределения x_i при условии родителей x_i, обозначаемых $\mathrm{P}a_{\mathcal{G}}(x_i)$:

$$p(\mathbf{x}) = \prod_i p(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i)). \tag{3.53}$$

На рис. 3.7 приведены пример ориентированного графа и представляемая им факторизация распределения вероятности.

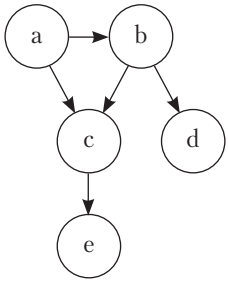

Рис. 3.7 ❖ Ориентированная графическая модель случайных величин a, b, c, d, e. Этот граф соответствует такой факторизации распределения вероятности:

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{a}, \mathrm{b})p(\mathrm{d} \mid \mathrm{b})p(\mathrm{e} \mid \mathrm{c}) \tag{3.54}$$

Эта графическая модель позволяет быстро выявить некоторые свойства распределения. Например, a и c взаимодействуют непосредственно, тогда как a и e – только косвенно, через c

В **неориентированных** моделях используются графы с неориентированными ребрами, они представляют разложение в произведение множества функций. В отличие от ориентированного случая, функции необязательно должны быть распределениями вероятности. Любое множество попарно соединенных вершин графа $\mathcal{G}$ называется кликой. Всякая клика $\mathcal{C}^{(i)}$ в неориентированной модели ассоциированна с фактором $\phi^{(i)}(\mathcal{C}^{(i)})$. Эти факторы – просто функции, а не распределения вероятности. Результат каждого фактора должен быть неотрицателен, но не требуется, чтобы сумма значений фактора или интеграл от него был равен 1, как в случае распределения вероятности.

Вероятность конфигурации случайных величин пропорциональна произведению всех факторов – комбинации, которые приводят к большим значениям факторов, более вероятны. Разумеется, нет никакой гарантии, что сумма произведений будет равна 1. Поэтому мы делим ее на нормировочную константу Z, определенную как сумма или интеграл по всем состояниям произведений функций ϕ, чтобы получить нормированное распределение вероятности:

$$p(\mathbf{x}) = \frac{1}{Z}\prod_i \phi^{(i)}(\mathcal{C}^{(i)}). \tag{3.55}$$

На рис. 3.8 приведены пример неориентированного графа и представляемая им факторизация распределения вероятности.

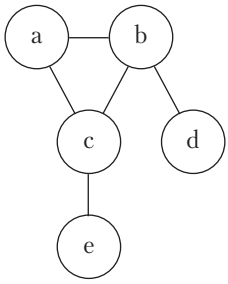

Рис. 3.8 ❖ Неориентированная, графическая модель случайных величин a, b, c, d, e. Этот граф соответствует такой факторизации распределения вероятности

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = \frac{1}{Z}\phi^{(1)}(\mathrm{a}, \mathrm{b}, \mathrm{c})\phi^{(2)}(\mathrm{b}, \mathrm{d})\phi^{(3)}(\mathrm{c}, \mathrm{e}). \tag{3.56}$$

Эта графическая модель позволяет быстро выявить некоторые свойства распределения. Например, a и c взаимодействуют непосредственно, тогда как a и e – только косвенно, через c.

Имейте в виду, что эти графические представления факторизаций – лишь язык описания распределений вероятности. Они не являются взаимно исключающими семействами распределений. Ориентированность или неориентированность – свойство не самого распределения вероятности, а конкретного **описания** распределения, и любое распределение можно описать обоими способами.

В частях I и II книги мы используем структурные вероятностные модели просто как язык, позволяющий описать, какие прямые вероятностные связи представлены различными алгоритмами машинного обучения. Более глубокое понимание структурных вероятностных моделей не понадобится вплоть до обсуждения тем для исследования в части III, где эти модели будут рассмотрены более детально.

В этой главе мы привели краткий обзор концепций теории вероятностей, имеющих прямое отношение к глубокому обучению. Осталось рассмотреть еще одну часть фундаментального математического аппарата: численные методы.

Глава 4

Численные методы

В алгоритмах машинного обучения обычно приходится выполнять много численных расчетов. Как правило, речь идет о применении методов, которые итеративно уточняют решение, а не ищут его аналитически по формуле. Типичные операции – оптимизация (нахождение значения, которое доставляет минимум или максимум некоторой функции) и решение системы линейных уравнений. Но даже само вычисление математической функции с помощью цифрового компьютера может оказаться трудной задачей, если в выражение функции входят вещественные числа, которые нельзя представить точно в памяти конечного размера.

4.1. Переполнение и потеря значимости

Фундаментальная сложность выполнения непрерывных математических операций на цифровом компьютере заключается в том, как представить бесконечно много вещественных чисел с помощью конечного числа комбинаций битов. Это означает, что почти для всех вещественных чисел производится некоторая аппроксимация и, следовательно, возникает ошибка округления. Такие ошибки составляют проблему, особенно если накапливаются в результате выполнения многих операций. Это может привести к тому, что теоретически правильный алгоритм, при проектировании которого не была предусмотрена минимизация накопления ошибок округления, на практике не работает.

Особенно неприятной разновидностью ошибок округления является **потеря значимости**. Это происходит, когда число, близкое к нулю, округляется до нуля. Многие функции ведут себя качественно различно, когда аргумент равен нулю, а не малому положительному числу. Например, обычно мы стремимся избежать деления на нуль (в одних программных средах это приводит к возбуждению исключения, а других – к возврату специального значения «не число») или взятия логарифма от нуля (обычно результат рассматривается как $-\infty$ и преобразуется в «не число» при попытке использования в последующих арифметических операциях).

Еще одна разновидность численных ошибок, приводящая к катастрофическим последствиям, – **переполнение**. Это происходит, когда абсолютная величина числа слишком велика и аппроксимируется как ∞ или $-\infty$. При последующих арифметических операциях такие бесконечные значения преобразуются в «не число».

Примером функции, которую необходимо защищать от потери значимости и переполнения, является **softmax**. Эта функция часто применяется для прогнозирования вероятностей, ассоциированных с категориальным распределением. Определяется она следующим образом:

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}. \tag{4.1}$$

Рассмотрим, что произойдет, когда все x_i равны некоторой константе c. Формально очевидно, что все компоненты результата должны быть равны $1/n$. Но в процессе вычислений так может не получиться, если абсолютная величина c очень велика. Если при этом c отрицательно, то при вычислении $\exp(c)$ произойдет потеря значимости. Тогда знаменатель softmax будет равен 0, и окончательный результат не определен. Если же c очень велико и положительно, то вычисление $\exp(c)$ приведет к переполнению, и результат выражения опять-таки будет не определен. Обе проблемы можно устранить, если вычислять $\text{softmax}(\boldsymbol{z})$, где $\boldsymbol{z} = \boldsymbol{x} - \max_i x_i$. Простые алгебраические выкладки показывают, что значение softmax не изменяется, если прибавить к входному вектору произвольный скаляр. После вычитания $\max_i x_i$ наибольший аргумент exp оказывается равен 0, что исключает возможность переполнения. Кроме того, по меньшей мере одно слагаемое в знаменателе равно 1, так что невозможна и потеря значимости в знаменателе, приводящая к делению на 0.

Но остается еще одна мелкая проблема. Из-за потери значимости в числителе все выражение может обратиться в нуль. Это означает, что если мы попытаемся вычислить $\log \text{softmax}(\boldsymbol{x})$, вызвав функцию softmax и передав результат функции log, то получим неверный результат $-\infty$. Поэтому мы должны вместо этого реализовать отдельную функцию, которая вычисляет log softmax численно устойчивым способом. Стабилизировать ее можно с помощью того же приема, что мы применили для стабилизации вычисления softmax.

Как правило, мы не будем явно оговаривать все детали вычислений, относящиеся к реализации описываемых в книге алгоритмов. Разработчики низкоуровневых библиотек должны помнить о проблемах численных расчетов при реализации алгоритмов глубокого обучения. Большинство читателей книги может просто полагаться на то, что низкоуровневые библиотеки позаботились о вычислительной устойчивости. Библиотека Theano (Bergstra et al., 2010; Bastien et al., 2012) – пример пакета, который автоматически обнаруживает и стабилизирует многие вычислительно неустойчивые выражения, часто встречающиеся в контексте глубокого обучения.

4.2. Плохая обусловленность

Под обусловленностью функции понимают скорость ее изменения в ответ на малые изменения аргументов. Функции, которые быстро изменяются при слабом возмущении аргументов, могут стать причиной проблем в научных расчетах, поскольку ошибки округления аргументов способны вызвать сильное изменение результата.

Рассмотрим функцию $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. Если матрица $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ имеет спектральное разложение, то ее **число обусловленности** равно

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \tag{4.2}$$

Это абсолютная величина отношения самого большого и самого маленького собственного значения. Если это число велико, то операция вычисления обратной матрицы особенно чувствительна к погрешности исходных данных.

Такая чувствительность – внутреннее свойство матрицы, а не результат ошибок округления при ее обращении. Если матрица плохо обусловлена, то уже имеющиеся погрешности усиливаются при умножении на истинно обратную к ней. На практике ошибка увеличивается еще больше из-за погрешностей, возникающих в процессе обращения.

4.3. Оптимизация градиентным методом

Большинство алгоритмов машинного обучения в том или ином виде включает оптимизацию, т. е. нахождение минимума или максимума функции $f(\boldsymbol{x})$ при изменении $\boldsymbol{x}$. Обычно задачу оптимизации формулируют в терминах нахождения минимума. Для нахождения максимума достаточно применить алгоритм минимизации к функции $-f(\boldsymbol{x})$.

Функция, для которой мы ищем минимум или максимум, называется **целевой функцией**, или **критерием**. Если речь идет о минимизации, то употребляют также термины **функция стоимости**, **функция потерь** или **функция ошибок**. В этой книге все эти термины используются как синонимы, хотя в других публикациях по машинному обучению некоторым из них приписывается специальный смысл.

Значение, доставляющее минимум или максимум функции, мы часто будем обозначать надстрочным индексом *, например: $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$.

Мы предполагаем, что читатель знаком с математическим анализом, но все же приведем краткий обзор понятий, относящихся к оптимизации.

Рассмотрим функцию $y = f(x)$, где x и y – вещественные числа. Ее производная обозначается $f'(x)$ или dy/dx. Производная $f'(x)$ определяет наклон $f(x)$ в точке x, т. е. коэффициент, на который нужно умножить малое изменение аргумента, чтобы получить соответственное изменение результата: $f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$.

Производная полезна для минимизации функции, потому что описывает, как изменить x, чтобы получить небольшое улучшение y. Например, мы знаем, что $f(x - \varepsilon\,\mathrm{sign}(f'(x)))$ меньше $f(x)$ при достаточно малом ε. Поэтому мы можем уменьшить $f(x)$, сдвигая x небольшими шагами в направлении, противоположном знаку производной. Этот метод называется **градиентным спуском** (Cauchy, 1847). Пример его применения показан на рис. 4.1.

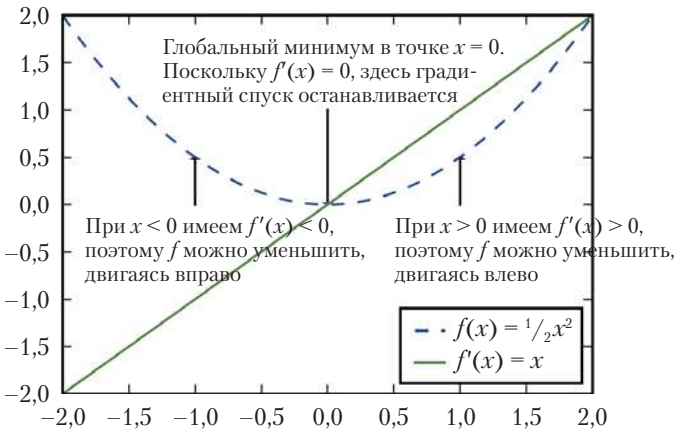

Рис. 4.1 ❖ Градиентный спуск. Иллюстрация применения метода градиентного спуска с использованием производной для перехода в точку минимума

Если $f'(x) = 0$, то производная не дает информации о том, в каком направлении смещаться. Точки, в которых $f'(x) = 0$, называются **критическими**, или **стационарными**. **Локальным минимумом** называется точка, в которой $f(x)$ меньше, чем во всех точках малой окрестности, поэтому уменьшить $f(x)$ путем изменения аргумента на небольшую величину невозможно. **Локальным максимумом** называется точка, в которой $f(x)$ больше, чем во всех точках малой окрестности, поэтому невозможно увеличить $f(x)$ путем изменения аргумента на небольшую величину. Некоторые критические точки не являются ни минимумами, ни максимумами. Они называются **седловыми точками**. На рис. 4.2 показаны примеры всех критических точек.

Рис. 4.2 ❖ Типы критических точек. Примеры трех типов критических точек в одномерном случае. Критической называется точка с нулевым угловым коэффициентом. Это может быть локальный минимум, в котором значение функции больше значений в окрестных точках, локальный максимум, в котором значение функции меньше значений в окрестных точках, или седловая точка, в которой значение функции может быть как больше, так и меньше значений в окрестных точках

Точка, в которой достигается абсолютно наименьшее значение $f(x)$, называется **глобальным минимумом**. У функции может быть один или несколько глобальных минимумов. Могут также существовать локальные минимумы, не являющиеся глобальными. В контексте глубокого обучения мы оптимизируем функции, у которых может быть много неоптимальных локальных минимумов, а также много седловых точек, окруженных очень плоскими участками. Все это затрудняет оптимизацию, особенно если речь идет о функциях нескольких переменных. Поэтому обычно считается достаточным найти малое значение f, формально не являющееся минимальным. Пример показан на рис. 4.3.

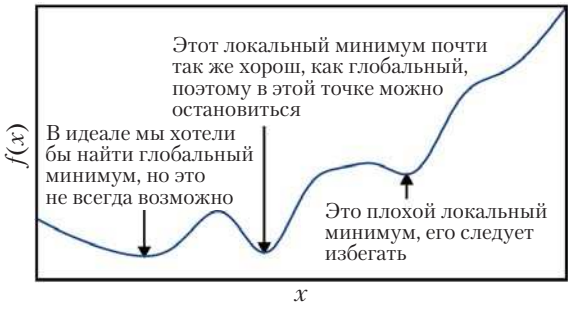

Рис. 4.3 ❖ Приближенная минимизация. Алгоритмы оптимизации могут не найти глобального минимума, если имеется несколько локальных минимумов или плато. В глубоком обучении мы обычно соглашаемся на такие решения, несмотря на то что это не настоящий минимум, при условии что они соответствуют действительно низким значениям функции стоимости

Часто приходится минимизировать функции нескольких переменных: $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. Чтобы понятие минимума имело смысл, результатом функции должен быть скаляр.

Для функций нескольких переменных следует ввести понятие **частной производной**. Частная производная $(\partial/\partial x_i)f(\boldsymbol{x})$ показывает, как изменяется f при изменении аргумента $\boldsymbol{x}$ только в одном направлении x_i. **Градиент** обобщает понятие производной на вектор: градиентом функции f называется вектор всех ее частных производных, он обозначается $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. i-м элементом градиента является частная производная f по x_i. В многомерном случае критическими называются точки, в которых все элементы градиента равны 0.

Производной по направлению в направлении единичного вектора $\boldsymbol{u}$ называется угловой коэффициент функции f в направлении u. Иначе говоря, производная по направлению – это производная функции $f(\boldsymbol{x} + \alpha\boldsymbol{u})$ по α, вычисленная при $\alpha = 0$. По правилу вычисления сложной производной $(\partial/\partial\alpha)f(\boldsymbol{x} + \alpha\boldsymbol{u})$ равно $\boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ при $\alpha = 0$.

Для минимизации f мы хотели бы найти направление, в котором f убывает быстрее всего. Это можно сделать с помощью производной по направлению:

$$\min_{\boldsymbol{u}, \boldsymbol{u}^\top \boldsymbol{u}=1} \boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{4.3}$$

$$= \min_{\boldsymbol{u}, \boldsymbol{u}^\top \boldsymbol{u}=1} \|\boldsymbol{u}\|_2 \|\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\|_2 \cos\theta. \tag{4.4}$$

где θ – угол между $\boldsymbol{u}$ и градиентом. Если подставить $\|\boldsymbol{u}\|_2 = 1$ и игнорировать множители, не зависящие $\boldsymbol{u}$, то остается $\min_{\boldsymbol{u}} \cos\theta$. Эта величина достигает минимума, когда $\boldsymbol{u}$ направлен противоположно градиенту. Иначе говоря, градиент указывает направление вверх, а отрицательный градиент – вниз. Чтобы уменьшить f, мы должны двигаться в направлении отрицательного градиента. Этот алгоритм называется **методом наискорейшего спуска**, или **градиентным спуском**.

В методе наискорейшего спуска предлагается выбирать новую точку

$$\boldsymbol{x}' = \boldsymbol{x} - \varepsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}), \tag{4.5}$$

где ε – **скорость обучения**, положительный скаляр, определяющий длину шага. Выбирать ε можно разными способами. Популярный подход – взять некоторую малую константу. Иногда удается найти размер шага, при котором производная по направлению обращается в нуль. Другой подход – вычислить $f(\boldsymbol{x} - \varepsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$ для нескольких значений ε и выбрать то, при котором целевая функция принимает наименьшее значение. Эта стратегия называется **линейным поиском**.

Метод наискорейшего спуска сходится, если все элементы градиента равны 0 (или, на практике, очень близки к нулю). В некоторых случаях можно избежать применения итеративного алгоритма и сразу перейти к критической точке, решив уравнение $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$ относительно $\boldsymbol{x}$.

Хотя градиентный спуск применим только к задачам оптимизации в непрерывных пространствах, идея выполнения малых шагов (аппроксимирующих наилучший малый шаг) для приближения к оптимальной конфигурации обобщается и на дискретные пространства. Поиск максимума целевой функции дискретных параметров называется методом **восхождения на вершину** (hill climbing) (Russel and Norvig, 2003).

4.3.1. Не только градиент: матрицы Якоби и Гессе

Иногда требуется найти все частные производные функции, аргументами и значением которой являются векторы. Матрица, содержащая такие частные производные, на-

зывается **матрицей Якоби**, или **якобианом**. Точнее, если имеется функция $\boldsymbol{f}: \mathbb{R}^m \longrightarrow \mathbb{R}^n$, то ее матрица Якоби $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ определяется как $J_{i,j} = (\partial / \partial x_j) f(\boldsymbol{x})_i$.

Иногда нас интересует также производная производной, или **вторая производная**. Например, для функции $f: \mathbb{R}^n \longrightarrow \mathbb{R}$ производная по x_i от производной f по x_j обозначается $(\partial^2 / \partial x_i \partial x_j) f$.

В одномерном случае $(d^2 f / dx^2) f$ обозначается также $f''(x)$. Вторая производная характеризует скорость изменения первой производной при изменении аргумента. Это важно, поскольку позволяет оценить, принесет ли шаг градиентного спуска ожидаемое улучшение. Можно считать, что вторая производная измеряет **кривизну**. Допустим, имеется квадратичная функция (многие возникающие на практике функции, хотя и не являются квадратичными, могут быть достаточно точно аппроксимированы ими, по крайней мере локально). Если вторая производная такой функции равна нулю, значит, кривизны нет. Это идеально плоская линия, значение которой можно предсказать, зная только градиент. Если градиент равен 1, то можно сделать шаг длины ε вдоль направления отрицательного градиента, и функция стоимости уменьшится на ε. Если вторая производная отрицательна, то функция изгибается вниз, поэтому функция стоимости уменьшится больше чем на ε. Наконец, если вторая производная положительна, то функция изгибается вверх, поэтому функция стоимости уменьшится меньше чем на ε. На рис. 4.4 показано, как от кривизны зависит связь между истинным значением функции стоимости и значением, предсказанным на основе анализа градиента.

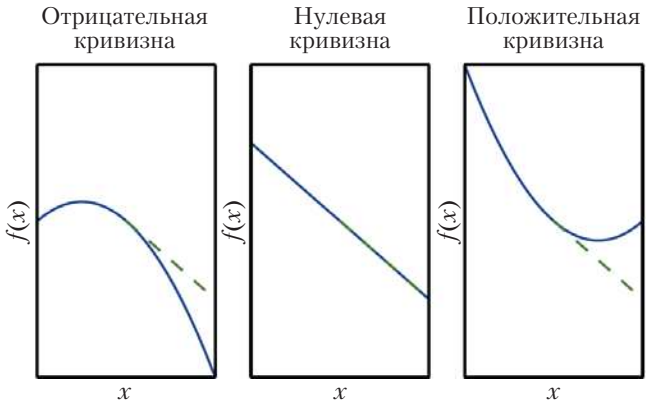

Рис. 4.4 ❖ Вторая производная определяет кривизну функции. Показаны квадратичные функции с разной кривизной. Штриховой линией обозначено значение функции стоимости, ожидаемое на основе анализа одного лишь градиента. Если кривизна отрицательна, то функция стоимости убывает быстрее, чем предсказывает градиент. Если кривизна равна нулю, то градиент правильно предсказывает значение. Если кривизна положительна, то функция стоимости убывает медленнее, чем предсказывает градиент, и в конечном счете начинает возрастать, поэтому если шаг выбран слишком большим, то можно случайно получить завышенное значение

В случае функции нескольких переменных вторых производных много. Их можно собрать в **матрицу Гессе**, или **гессиан**. Матрица Гессе $\boldsymbol{H}(f)(\boldsymbol{x})$ определяется следующим образом:

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}). \tag{4.6}$$

Можно также сказать, что матрица Гессе является якобианом градиента.

Всюду, где вторые частные производные непрерывны, операторы дифференцирования коммутативны, т. е. их можно менять местами:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\boldsymbol{x}). \tag{4.7}$$

Это означает, что $H_{i,j} = H_{j,i}$, т. е. в таких точках матрица Гессе симметрична. Для большинства функций, встречающихся в глубоком обучении, матрица Гессе является вещественной и симметричной почти всюду. А раз так, то мы можем найти множество вещественных собственных значений и ортогональный базис собственных векторов. Вторая производная в направлении единичного вектора $\boldsymbol{d}$, по определению, равна $\boldsymbol{d}^\top \boldsymbol{H}\boldsymbol{d}$. Если $\boldsymbol{d}$ – собственный вектор $\boldsymbol{H}$, то вторая производная в этом направлении равна соответствующему собственному значению. Для других направлений $\boldsymbol{d}$ вторая производная по направлению равна взвешенному среднему всех собственных значений с весами от 0 до 1, причем чем меньше угол между собственным вектором и $\boldsymbol{d}$, тем больше вес этого вектора. Максимальное собственное значение определяет максимальную вторую производную, а минимальное – минимальную. Вторая производная по направлению дает информацию об ожидаемом качестве шага градиентного спуска. Можно аппроксимировать функцию $f(\boldsymbol{x})$ в окрестности точки $\boldsymbol{x}^{(0)}$, оставив только члены не выше второго порядка в ее разложении в ряд Тейлора:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + {}^1\!/_2 (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)}), \tag{4.8}$$

где $\boldsymbol{g}$ – градиент, а $\boldsymbol{H}$ – гессиан в точке $\boldsymbol{x}^{(0)}$. Если скорость обучения равна ε, то новая точка $\boldsymbol{x}$ определяется по формуле $\boldsymbol{x}^{(0)} - \varepsilon \boldsymbol{g}$. Подставляя в приведенную выше формулу, получаем:

$$f(\boldsymbol{x}^{(0)} - \varepsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \varepsilon \boldsymbol{g}^\top \boldsymbol{g} + {}^1\!/_2 \varepsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}. \tag{4.9}$$

В этой формуле три члена: исходное значение функции, ожидаемое улучшение, обусловленное наклоном функции, и поправка на кривизну функции. Если последний член слишком велик, то шаг градиентного спуска может в действительности привести к подъему. Если $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ равно нулю или отрицательно, то аппроксимация рядом Тейлора предсказывает, что при постоянном увеличении ε функция f будет постоянно убывать. На практике ряд Тейлора редко дает точную аппроксимацию для больших ε, поэтому при выборе значения ε приходится прибегать к различным эвристическим соображениям. Если $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ положительно, то, решая уравнение, находим оптимальную величину шага, при которой аппроксимация функции рядом Тейлора убывает в наибольшей степени:

$$\varepsilon^* = \frac{\boldsymbol{g}^\top \boldsymbol{g}}{\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}}. \tag{4.10}$$

В худшем случае, когда $\boldsymbol{g}$ совпадает по направлению с собственным вектором $\boldsymbol{H}$, соответствующим максимальному собственному значению $\lambda_{\max}$, эта оптимальная величина шага равна $1/\lambda_{\max}$. Следовательно, если минимизируемую функцию вообще

можно хорошо аппроксимировать квадратичной, собственные значения матрицы Гессе определяют масштаб скорости обучения.

Вторую производную можно использовать, чтобы узнать, является ли критическая точка локальным максимумом, локальным минимумом или седловой точкой. Напомним, что в критической точке $f'(x) = 0$. Если вторая производная $f''(x) > 0$, то первая производная $f'(x)$ возрастает при сдвиге вправо и убывает при сдвиге влево, т. е. $f'(x - \varepsilon) < 0$ и $f'(x + \varepsilon) > 0$ для достаточно малых ε. Иными словами, когда мы смещаемся вправо, угловой коэффициент указывает на подъем с правой стороны, а при смещении влево – на подъем с левой стороны. Следовательно, если $f'(x) = 0$ и $f''(x) > 0$, мы заключаем, что x – локальный минимум. Аналогично, если $f'(x) = 0$ и $f''(x) < 0$, то x – локальный максимум. Это так называемая **проверка по второй производной**. К сожалению, если $f''(x) = 0$, то эта проверка не дает однозначного результата. В таком случае x может быть седловой точкой или находиться на плоском участке.

В многомерном случае необходимо исследовать все вторые производные функции. С помощью спектрального разложения матрицы Гессе мы можем обобщить проверку по второй производной на многомерный случай. В критической точке $\nabla_x f(\boldsymbol{x}) = 0$, поэтому путем анализа собственных значений гессиана можно узнать, является ли эта точка локальным максимумом, локальным минимумом или седловой точкой. Если матрица Гессе положительно определенная (все ее собственные значения положительны), то это локальный минимум. В этом можно убедиться, заметив, что вторая производная по любому направлению должна быть положительна, и сославшись на проверку по второй производной в одномерном случае. Аналогично, если матрица Гессе отрицательно определенная (все собственные значения отрицательны), точка является локальным максимумом. В многомерном случае иногда удается найти свидетельства в пользу седловой точки. Если имеется хотя бы одно положительное и хотя бы одно отрицательное собственное значение, то мы знаем, что $\boldsymbol{x}$ является локальным максимумом в одном сечении f и локальным минимумом в другом. Пример приведен на рис. 4.5. Наконец, проверка по второй производной в многомерном случае может не давать однозначного результата, как и в одномерном. Так бывает, когда все ненулевые собственные значения одного знака, но имеется хотя бы одно нулевое. Неоднозначность возникает из-за недостаточной информативности одномерной проверки второй производной в сечении, соответствующем нулевому собственному значению.

В многомерном случае в одной точке вторые производные по каждому направлению различны. Число обусловленности матрицы Гессе в точке измеряет степень различия вторых производных. Если число обусловленности велико, то градиентный спуск будет работать плохо. Это объясняется тем, что в одном направлении производная растет быстро, а в другом медленно. Метод градиентного спуска не в курсе этого различия, поэтому не знает, что предпочтительным направлением для исследования является то, в котором производная дольше остается отрицательной. Из-за плохого числа обусловленности трудно выбрать хорошую величину шага. Шаг должен быть достаточно малым, что не пропустить минимум и подниматься вверх во всех направлениях, где кривизна строго положительна. Но обычно это означает, что шаг слишком мал для заметного продвижения в направлениях с меньшей кривизной. Пример приведен на рис. 4.6.

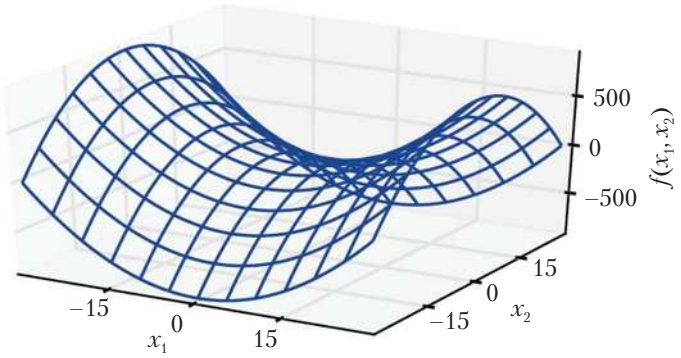

Рис. 4.5 ❖ Седловая точка, в которой присутствует как положительная, так и отрицательная кривизна. Представлен график функции $f(\boldsymbol{x}) = x_1^2 - x_2^2$. Вдоль оси x_1 функция изгибается вверх. Направление этой оси – собственный вектор матрицы Гессе с положительным собственным значением. Вдоль оси x_2 функция изгибается вниз. Это направление собственного вектора матрицы Гессе с отрицательным собственным значением. Название «седловая точка» связано с тем, что эта поверхность напоминает седло. Это хрестоматийный пример функции с седловой точкой. В многомерном случае для наличия седловой точки необязательно, чтобы собственное значение было равно 0; необходимо лишь, чтобы существовали как положительные, так и отрицательные собственные значения. В седловой точке, соответствующей собственным значениям разных знаков, в одном сечении достигается локальный максимум, а в другом – локальный минимум

Эту проблему можно решить, используя матрицу Гессе для управления поиском. Простейший метод такого рода известен под названием **метода Ньютона**. Он основан на разложении $f(\boldsymbol{x})$ в ряд Тейлора с точностью до членов второго порядка в окрестности точки $\boldsymbol{x}^{(0)}$:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) + {}^1\!/_2 (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})(\boldsymbol{x} - \boldsymbol{x}^{(0)}). \tag{4.11}$$

Из этого уравнения находим критическую точку функции:

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})^{-1} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}). \tag{4.12}$$

Если f – положительно определенная квадратичная функция, то метод Ньютона состоит в однократном применении уравнения (4.12) для непосредственного нахождения точки минимума. Если функция f не квадратичная, но может быть локально аппроксимирована положительно определенной квадратичной функцией, то уравнение (4.12) нужно применить несколько раз. Итеративное уточнение аппроксимации с нахождением минимума аппроксимирующей функции позволяет достичь критической точки гораздо быстрее, чем метод градиентного спуска. Это полезное свойство вблизи локального минимума, но оно может оказаться опасным в окрестности седловой точки. В разделе 8.2.3 мы увидим, что метод Ньютона годится, только если ближайшая критическая точка является минимумом (все собственные значения матрицы Гессе положительны), тогда как метод градиентного спуска не притягивается к седловой точке, если только на нее не указывает градиент.

Алгоритмы оптимизации, основанные только на градиенте, в частности метод градиентного спуска, называются **алгоритмами оптимизации первого порядка**. Если

учитывается также гессиан, как в методе Ньютона, то это **алгоритм оптимизации второго порядка** (Nocedal and Wright, 2006).

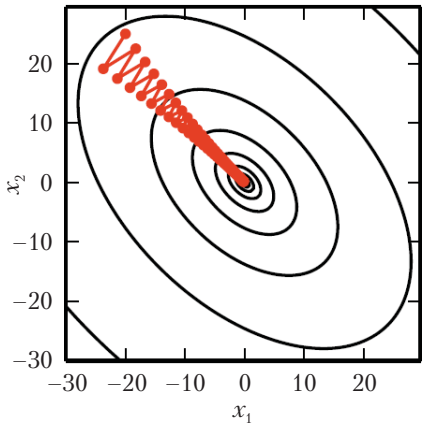

Рис. 4.6 ❖ Метод градиентного спуска не использует информацию о кривизне, содержащуюся в гессиане. Здесь градиентный спуск применяется для минимизации квадратичной функции $f(\boldsymbol{x})$, для которой число обусловленности гессиана равно 5. Это означает, что величина кривизны в направлениях наибольшей и наименьшей кривизны различается в пять раз. В данном случае направление наибольшей кривизны совпадает с вектором $[1, 1]^\top$, а наименьшей – с вектором $[1, -1]^\top$. Красной линией показан путь, которым следует метод градиентного спуска. Эта вытянутая квадратичная функция напоминает длинный каньон. Градиентный спуск бесполезно тратит много времени, раз за разом спускаясь по стенкам каньона, поскольку это направления наискорейшего спуска. Так как шаг слишком большой, мы проскакиваем мимо дна каньона и вынуждены спускаться по противоположной стенке на следующей итерации. Большое положительное значение матрицы Гессе, соответствующее собственному вектору в этом направлении, подсказывает, что производная по этому направлению быстро возрастает, поэтому алгоритм оптимизации мог бы воспользоваться гессианом и понять, что в данном случае производить поиск в направлении наискорейшего спуска не стоит

Алгоритмы оптимизации, применяемые в большинстве случаев, рассматриваемых в этой книге, годятся для широкого спектра функций, но не дают почти никаких гарантий. Отсутствие гарантий связано с тем, что функции, используемые в глубоком обучении, чрезвычайно сложны. Во многих других областях принято разрабатывать алгоритмы оптимизации для ограниченного семейства функций.

В контексте глубокого обучения иногда можно получить некоторые гарантии, если ограничиться функциями, которые либо удовлетворяют **условию Липшица**, либо имеют производные, удовлетворяющие этому условию. Говорят, что функция f удовлетворяет условию Липшица, если скорость ее изменения ограничена некоторой константой $\mathcal{L}$, которая называется постоянной Липшица:

$$\forall \boldsymbol{x}, \forall \boldsymbol{y}, |f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq \mathcal{L}\|\boldsymbol{x} - \boldsymbol{y}\|_2. \tag{4.13}$$

Это свойство полезно, т. к. позволяет количественно выразить предположение о том, что небольшое изменение аргументов, как, например, в алгоритме градиентно-

го спуска, приводит к небольшому изменению функции. Условие Липшица является довольно слабым ограничением, и многие задачи оптимизации в глубоком обучении можно свести к этому случаю путем сравнительно небольшой модификации.

Пожалуй, самым успешным примером специализированной оптимизации является **выпуклая оптимизация**. Алгоритмы выпуклой оптимизации могут дать куда большие гарантии ценой более строгих ограничений. Они применимы только к выпуклым функциям – таким, для которых матрица Гессе является всюду положительно полуопределенной. Такие функции хорошо себя ведут, потому что не могут иметь седловых точек, а все локальные минимумы обязательно глобальны. Однако большинство задач глубокого обучения трудно выразить в терминах выпуклой оптимизации. Выпуклая оптимизация применяется только в качестве подпрограммы в некоторых алгоритмах глубокого обучения. Идеи, заимствованные из выпуклой оптимизации, бывают полезны для доказательства сходимости алгоритмов, но в общем случае ценность выпуклой оптимизации в контексте глубокого обучения невелика. Дополнительные сведения о выпуклой оптимизации см. в работах Boyd and Vandenberghe (2004) или Rockafellar (1997).

4.4. Оптимизация с ограничениями

Иногда задача состоит в том, чтобы найти максимум или минимум функции $f(\boldsymbol{x})$ не во всей области допустимых значений $\boldsymbol{x}$, а лишь в некотором ее подмножестве $\mathbb{S}$. Такая задача называется **оптимизацией с ограничениями**, или **ограниченной оптимизацией**. В этом случае точки множества $\mathbb{S}$ называются **допустимыми**.

Часто нас интересует решение, которое в некотором смысле мало. В таких случаях обычно налагается ограничение на норму, например $\|\boldsymbol{x}\| \leq 1$.

Для решения задачи оптимизации с ограничениями можно просто изменить метод градиентного спуска, учтя ограничение. Если используется небольшой постоянный шаг ε, то можно выполнить шаги градиентного спуска, а затем спроецировать результаты обратно в $\mathbb{S}$. При использовании линейного поиска можно искать только с такими шагами, при которых новая точка $\boldsymbol{x}$ оказывается допустимой, или же проецировать каждую точку на прямой обратно в область ограничения. Там, где возможно, эффективность этого метода можно повысить, проецируя градиент на касательное пространство к допустимой области, перед тем как делать шаг или начинать линейный поиск (Rosen, 1960).

Более сложный подход – сформулировать другую задачу оптимизации без ограничений, решение которой можно преобразовать в решение исходной задачи с ограничениями. Например, если мы хотим минимизировать $f(\boldsymbol{x})$ для $\boldsymbol{x} \in \mathbb{R}^2$ с ограничением – $\boldsymbol{x}$ должно иметь строго единичную норму L^2, то можно вместо этого минимизировать функцию $g(\theta) = f([\cos\theta, \sin\theta]^\top)$ относительно θ, а затем вернуть $[\cos\theta, \sin\theta]$ в качестве решения исходной задачи. Этот подход требует изобретательности; для каждой задачи оптимизации приходится придумывать отдельное преобразование.

Метод Каруша–Куна–Таккера (ККТ)[1] предлагает очень общий подход к решению задач оптимизации. Вводится новая функция, называемая **обобщенным лагранжианом**, или **обобщенной функцией Лагранжа**.

---

[1] Метод ККТ является обобщением метода **множителей Лагранжа**, который допускает только ограничения типа равенств.

Чтобы определить лагранжиан, мы сначала должны описать $\mathbb{S}$ с помощью равенств и неравенств, т. е. m функций $g^{(i)}$ и n функций $h^{(j)}$ – таких, что $\mathbb{S} = \{x \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0$ и $\forall j, h^{(j)}(\boldsymbol{x}) \leq 0\}$. Условия, содержащие функции $g^{(i)}$, называются **ограничениями типа равенств**, а содержащие функции $h^{(i)}$ – **ограничениями типа неравенств**.

Для каждого ограничения вводятся новые переменные λ_i и α_j, называемые множителями ККТ. Обобщенный лагранжиан записывается в виде:

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.14)$$

Теперь задачу минимизации с ограничениями можно решить, подвергнув обобщенный лагранжиан оптимизации без ограничений. Если существует хотя бы одна допустимая точка и $f(\boldsymbol{x})$ не может принимать значение 1, то

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \quad (4.15)$$

обладает такой же целевой функцией и множеством оптимальных точек $\boldsymbol{x}$, что и

$$\min_{\boldsymbol{x} \in \mathbb{S}} f(\boldsymbol{x}). \quad (4.16)$$

Это следует из того, что если ограничения удовлетворяются, то

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}), \quad (4.17)$$

а если некоторое ограничение нарушается, то

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty. \quad (4.18)$$

Эти свойства гарантируют, что никакая недопустимая точка не может быть оптимальной, а величина оптимума на множестве допустимых точек не изменяется.

Чтобы выполнить максимизацию с ограничениями, мы можем построить обобщенный лагранжиан $-f(\boldsymbol{x})$, который приводит к такой задаче оптимизации:

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} -f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.19)$$

Можно также свести задачу к задаче максимизации во внешнем цикле:

$$\max_{\boldsymbol{x}} \min_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) - \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.20)$$

Знак члена, содержащего ограничения типа равенств, не имеет значения; можно выбрать как плюс, так и минус, потому что алгоритм оптимизации волен выбирать любые знаки коэффициентов λ_i.

Ограничения типа неравенств особенно интересны. Мы говорим, что ограничение $h^{(i)}(\boldsymbol{x})$ **активно**, если $h^{(i)}(\boldsymbol{x}^*) = 0$. Если ограничение не активно, то решение задачи, найденное при наличии этого ограничения, останется, по меньшей мере, локальным решением, если ограничение снять. Может случиться, что неактивное ограничение исключает другие решения. Например, в выпуклой задаче, где имеется целая область глобально оптимальных точек (широкая плоская область с одинаковой стоимостью), некоторое подмножество этой области может исключаться ограничениями. А в невыпуклой задаче, возможно, удалось бы найти лучшие локальные стационарные точки, но они исключаются ограничениями, неактивными в точке сходимости. Тем не менее точка сходимости остается стационарной точкой вне зависимости от

того, включены или нет неактивные ограничения. Поскольку значение неактивной функции $h^{(i)}$ отрицательно, в решении задачи $\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ коэффициент $\alpha_i = 0$. Таким образом, в решении $\boldsymbol{\alpha} \odot \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$. Иначе говоря, для всех i мы знаем, что хотя бы одно из ограничений $-\alpha_i \geq 0$ или $h^{(i)}(\boldsymbol{x}) \leq 0$ – должно быть активно для решения. Интуитивно эту мысль можно выразить, сказав, что решение либо находится на границе, определяемой неравенством, и тогда мы должны использовать его множитель ККТ, либо неравенство вообще не влияет на решение, и тогда его множитель ККТ обнуляется.

У оптимальных точек задач оптимизации с ограничениями есть простой набор свойств, называемых условиями Каруша–Куна–Таккера (Karush, 1939; Kuhn and Tucker, 1951). Это необходимые, но недостаточные условия оптимальности точки.

- Градиент обобщенного лагранжиана равен нулю.
- Все ограничения на $\boldsymbol{x}$ и на множители ККТ удовлетворяются.
- Ограничения типа неравенств обладают свойством «дополняющей нежесткости»: $\boldsymbol{\alpha} \odot \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$.

Дополнительные сведения о методе ККТ см. в работе Nocedal and Wright (2006).

4.5. Пример: линейный метод наименьших квадратов

Требуется найти значение $\boldsymbol{x}$, минимизирующее функцию

$$f(\boldsymbol{x}) = {}^1\!/_2 \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2. \tag{4.21}$$

В линейной алгебре рассматриваются специализированные алгоритмы для эффективного решения этой задачи, но мы покажем, как решить ее методами градиентной оптимизации.

Сначала получим выражение для градиента:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b}. \tag{4.22}$$

Затем будем небольшими шагами спускаться вдоль направления градиента. Детали описаны в алгоритме 4.1.

Алгоритм 4.1. Алгоритм минимизации $f(\boldsymbol{x}) = {}^1\!/_2 \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$ относительно $\boldsymbol{x}$ методом градиентного спуска, начиная с произвольного значения $\boldsymbol{x}$

Взять в качестве величины шага (ε) и допуска (δ) небольшие положительные числа.
while $\|\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b}\|_2 > \delta$ **do**
  $\boldsymbol{x} \leftarrow \boldsymbol{x} - \varepsilon(\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b})$
end while

Эту задачу можно решить также методом Ньютона. Поскольку истинная функция квадратичная, то квадратичная аппроксимация в методе Ньютона является точной, и алгоритм сходится к глобальному минимуму за один шаг.

Теперь предположим, что требуется минимизировать ту же функцию, но при наличии ограничения $\boldsymbol{x}^\top\boldsymbol{x} \leq 1$. Введем в рассмотрение лагранжиан:

$$L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda(\boldsymbol{x}^\top\boldsymbol{x} - 1)\,. \tag{4.23}$$

Теперь можно решать задачу:

$$\min_{\boldsymbol{x}} \max_{\lambda,\lambda\geq 0} L(\boldsymbol{x}, \lambda). \tag{4.24}$$

Решение задачи без ограничений с наименьшей нормой можно найти с помощью псевдообратной матрицы Мура–Пенроуза: $\boldsymbol{x} = \boldsymbol{A}^{+}\boldsymbol{b}$. Если эта точка допустима, то мы нашли решение задачи с ограничениями. В противном случае нужно найти решение, для которого ограничение активно. Продифференцировав лагранжиан по $\boldsymbol{x}$, получаем уравнение:

$$\boldsymbol{A}^{\top}\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^{\top}\boldsymbol{b} + 2\lambda\boldsymbol{x} = 0. \tag{4.25}$$

Отсюда следует, что решение нужно искать в виде:

$$\boldsymbol{x} = (\boldsymbol{A}^{\top}\boldsymbol{A} + 2\lambda\boldsymbol{I})^{-1}\boldsymbol{A}^{\top}\boldsymbol{b}. \tag{4.26}$$

Абсолютную величину λ следует выбирать так, чтобы результат удовлетворял ограничению. Чтобы найти такое значение, можно выполнить градиентный подъем по λ. Для этого заметим, что

$$\frac{\partial}{\partial\lambda} L(\boldsymbol{x}, \lambda) = \boldsymbol{x}^{\top}\boldsymbol{x} - 1. \tag{4.27}$$

Когда норма $\boldsymbol{x}$ превышает 1, эта производная положительна, поэтому, чтобы подниматься вверх в направлении производной и увеличивать лагранжиан относительно λ, мы увеличиваем λ. Поскольку штрафной коэффициент при $\boldsymbol{x}^{\top}\boldsymbol{x}$ увеличился, разрешение линейного уравнения относительно $\boldsymbol{x}$ теперь даст решение с меньшей нормой. Процесс решения линейного уравнения и корректировки λ продолжается до тех пор, пока норма $\boldsymbol{x}$ не окажется допустимой и производная по λ не станет равна 0.

На этом мы завершаем обзор математического аппарата, который понадобится для разработки алгоритмов машинного обучения. Все готово к построению и анализу полноценных систем обучения.

Глава 5

Основы машинного обучения

Глубокое обучение – частный случай машинного обучения. Чтобы хорошо понимать глубокое обучение, нужно как следует усвоить основные принципы машинного обучения. В этой главе кратко излагаются самые важные общие принципы машинного обучения, которые найдут применение в остальной части книги. Читателям, совсем незнакомым с предметом или желающим получить более широкое представление о нем, рекомендуем учебники, в которых фундаментальные основы рассматриваются более подробно, например Murphy (2012) или Bishop (2006). Если вы уже знакомы с машинным обучением, можете сразу перейти к разделу 5.11, где обсуждаются некоторые взгляды на традиционное машинное обучение, оказавшие значительное влияние на разработку алгоритмов глубокого обучения.

Начнем с определения алгоритма обучения и приведем пример: алгоритм линейной регрессии. Затем опишем, чем проблема соответствия обучающим данным отличается от проблемы поиска закономерностей, которые обобщаются на новые данные. У большинства алгоритмов машинного обучения есть настройки, которые называются *гиперпараметрами* и должны определяться вне самого алгоритма обучения; мы обсудим, как задавать их с помощью дополнительных данных. Машинное обучение по сути своей является вариантом прикладной статистики, когда особый упор делается на использовании компьютеров для статистического оценивания сложных функций, а меньше внимания уделяется определению доверительных интервалов для этих функций. Поэтому мы представим два основных подхода к статистике: частотные оценки и байесовский вывод. Большинство алгоритмов машинного обучения можно отнести к одной из двух категорий: с учителем и без учителя; мы опишем, что это значит, и приведем примеры простых алгоритмов из каждой категории. Большинство алгоритмов глубокого обучения основано на алгоритме оптимизации, который называется стохастическим градиентным спуском. Мы покажем, как алгоритм глубокого обучения собирается из составных частей: алгоритма оптимизации, функции стоимости, модели и набора данных. Наконец, в разделе 5.11 мы опишем некоторые факторы, ограничивающие способность традиционного машинного обучения к обобщению. Именно стремление преодолеть эти проблемы и стало побудительным мотивом к разработке алгоритмов глубокого обучения.

5.1. Алгоритмы обучения

Алгоритм машинного обучения – это алгоритм, способный обучаться на данных. Но что мы понимаем под обучением? В работе Mitchell (1997) дано такое лаконичное определение: «Говорят, что компьютерная программа обучается на опыте E относительно некоторого класса задач T и меры качества P, если качество на задачах из T, измеренное с помощью P, возрастает с ростом опыта E». Можно представить себе широкое разнообразие опыта E, задач T и мер качества P, и в этой книге мы не будем пытаться формально определить, что можно использовать в качестве каждой из этих сущностей. Вместо этого в следующих разделах мы приведем интуитивно понятные описания и примеры различных задач, мер качества и видов опыта, пригодных для конструирования алгоритмов машинного обучения.

5.1.1. Задача *T*

Машинное обучение позволяет справляться с задачами, которые слишком сложны для решения с помощью фиксированных программ, спроектированных и написанных людьми. С научной и философской точки зрения, машинное обучение интересно тем, что чем лучше мы понимаем его, тем глубже становится наше понимание принципов, лежащих в основе разума.

Говоря о «задаче», сразу отметим, что сам процесс обучения задачей не является. Обучение – это средство, благодаря которому мы получаем возможность выполнить задачу. Например, если мы хотим, чтобы робот мог ходить, то хождение – это задача. Мы можем запрограммировать робота на обучение хождению или попробовать вручную написать программу, которая описывает, как надо ходить.

Задачи машинного обучения обычно описываются в терминах того, как система машинного обучения должна обрабатывать **пример**. Примером является набор **признаков**, полученных в результате количественного измерения некоторого объекта или события, которые система должна научиться обрабатывать. Как правило, пример представляется в виде вектора $\boldsymbol{x} \in \mathbb{R}^n$, каждый элемент которого – признак. Так, признаками изображения обычно являются значения его пикселей.

С помощью машинного обучения можно решать разнообразные задачи. Перечислим наиболее типичные.

- **Классификация.** В задачах этого типа программа должна ответить, какой из k категорий принадлежит некоторый пример. Для решения этой задачи алгоритм обучения обычно просят породить функцию $f : \mathbb{R}^n \rightarrow \{1, ..., k\}$. Если $y = f(\boldsymbol{x})$, то модель относит входной пример, описываемый вектором $\boldsymbol{x}$, к категории с числовым кодом y. Существуют и другие варианты задачи классификации, например когда f – распределение вероятности принадлежности к классам. Примером задачи классификации является распознавание объектов, когда на вход подается изображение (обычно представленное множеством значений яркости пикселей), а на выходе получается числовой код, который идентифицирует присутствующий в изображении объект. Например, робот Willow Garage PR2 может исполнять функции официанта: он умеет распознавать различные напитки и по команде приносить их людям (Goodfellow et al., 2010). В настоящее время распознавание объектов лучше всего производится методами глубокого обучения (Krizhevsky et al., 2012; Ioffe and Szegedy, 2015). Распознавание объектов основано на той же базовой технологии, которая позволяет компьютерам

распознавать лица (Taigman et al., 2014). Это можно использовать для автоматической пометки людей в коллекциях фотографий и для организации более естественного взаимодействия компьютера с пользователями.

- **Классификация при отсутствии некоторых данных.** Классификация осложняется, если нет гарантии, что программа получает во входном векторе результаты всех измерений. Чтобы решить задачу классификации, алгоритм обучения должен определить всего одну функцию, отображающую входной вектор на код категории. Но если часть входных данных отсутствует, то алгоритм должен обучить набор функций. Каждая функция соответствует классификации $\boldsymbol{x}$ в условиях, когда отсутствуют различные подмножества данных. Один из способов эффективно определить такое большое множество функций – обучить распределение вероятности всех релевантных величин, а затем решать задачу классификации, вычисляя маргинальные распределения отсутствующих величин. Если на вход подается n величин, то существует 2^n различных функций классификации для каждого возможного набора отсутствующих данных, однако программе нужно обучить только одну функцию, описывающую совместное распределение вероятности. Пример глубокой вероятностной модели, применимой к такой задаче, приведен в работе Goodfellow et al. (2013b). Многие другие задачи, описанные в этом разделе, также допускают обобщение на случай отсутствующих данных; классификация при отсутствии некоторых данных – лишь один пример того, на что способно машинное обучение.
- **Регрессия.** В этой задаче программа должна предсказать числовое значение по входным данным. Для ее решения алгоритм обучения просят породить функцию $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Регрессия отличается классификации форматом выхода. Примером может служить прогнозирование суммы претензии, на которую будет претендовать застрахованный (это нужно для определения размера страховой премии), или прогнозирование будущей стоимости ценных бумаг. Такого рода прогнозы применяются также в алгоритмическом трейдинге.
- **Транскрипция.** В задачах такого типа системе машинного обучения предлагается проанализировать неструктурированное представление некоторых данных и преобразовать его в дискретную текстовую форму. Например, программе распознавания текста предъявляется фотография текста, а она должна вернуть текст в виде последовательности символов (скажем, в кодировке ASCII или Unicode). В системе Google Street View глубокое обучение используется для подобной обработки табличек с адресами домов (Goodfellow et al., 2014d). Другой пример – распознавание речи, когда программе предъявляется аудиосигнал, а она выводит последовательность символов или идентификаторов произнесенных слов. Глубокое обучение – важнейший компонент современных систем распознавания речи, используемых в таких компаниях, как Microsoft, IBM и Google (Hinton et al., 2012b).
- **Машинный перевод.** В этой задаче входные данные – последовательность символов на одном языке, а программа должна преобразовать ее в последовательность символов на другом языке. Обычно такие системы применяются к естественным языкам, скажем, для перевода с английского на французский. В последнее время глубокое обучение стало проникать и в эту область (Sutskever et al., 2014; Bahdanau et al., 2015).

- **Структурный вывод.** Под структурным выводом понимается любая задача, в которой на выходе порождается вектор (или иная структура, содержащая несколько значений), между элементами которого существуют важные связи. Это широкая категория, в которую входят, в частности, задачи транскрипции и перевода. Еще одна задача – грамматический разбор, т. е. преобразование предложения на естественном языке в дерево, описывающее его грамматическую структуру; узлы такого дерева снабжены метками «глагол», «существительное», «наречие» и т. д. Пример применения глубокого обучения к задаче грамматического разбора имеется в работе Collobert (2011). Можно также рассмотреть пиксельную сегментацию изображения, когда программа должна отнести каждый пиксель к определенной категории. Например, глубокое обучение можно использовать для аннотирования дорог на аэрофотоснимках (Mnih and Hinton, 2010). Формат выхода не всегда так точно повторяет формат ввода, как в задачах, сводящихся к аннотированию. Например, в задаче подписывания изображений программе предъявляется изображение, а она выводит его описание в виде предложения на естественном языке (Kiros et al., 2014a,b; Mao et al., 2015; Vinyals et al., 2015b; Donahue et al., 2014; Karpathy and Li, 2015; Fang et al., 2015; Xu et al., 2015). Название «структурный вывод» отражает тот факт, что программа должна вывести несколько тесно связанных между собой значений. Так, слова, порождаемые программой подписывания изображений, должны образовывать допустимое предложение.
- **Обнаружение аномалий.** В этой задаче программа анализирует множество событий или объектов и помечает некоторые из них как нетипичные. Примером может служить обнаружение мошенничества с кредитными картами. Благодаря моделированию покупательских привычек компания-эмитент кредитных карт может обнаружить аномальное использование карты. Покупки человека, укравшего вашу карту или информацию о ней, зачастую характеризуются не таким распределением вероятности, как ваши. Обнаружив аномальную покупку, компания сможет предотвратить мошенничество, заблокировав счет. Обзор методов обнаружения аномалий приведен в работе Chandola et al. (2009).
- **Синтез и выборка.** В задачах этого типа алгоритм машинного обучения должен генерировать новые примеры, похожие на обучающие данные. Синтез и выборка методами машинного обучения полезны в мультимедийных приложениях, когда генерирование больших объемов данных вручную обходится дорого, скучно или занимает слишком много времени. Например, видеоигры умеют автоматически генерировать текстуры для больших объектов или ландшафтов, не заставляя художника вручную помечать каждый пиксель (Luo et al., 2013). В некоторых случаях мы хотим, чтобы процедура выборки или синтеза генерировала определенный выход по заданному входу. Например, в задаче синтеза речи программе передается написанная фраза, а она должна выдать аудиосигнал, содержащий ее же в произнесенном виде. Это разновидность структурного вывода с дополнительной особенностью: для заданного входа не существует единственно правильного выхода, мы специально допускаем большую вариативность выхода, чтобы речевой сигнал звучал естественно и реалистично.
- **Подстановка отсутствующих значений.** В этом случае алгоритму машинного обучения предъявляется новый пример $\boldsymbol{x} \in \mathbb{R}^n$, в котором некоторые

элементы x_i отсутствуют. Алгоритм должен спрогнозировать значения отсутствующих элементов.

- **Шумоподавление.** В этой задаче алгоритму машинного обучения предъявляется искаженный помехами пример $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$, полученный из чистого примера $\boldsymbol{x} \in \mathbb{R}^n$ в результате неизвестного процесса искажения. Алгоритм должен восстановить чистый пример $\boldsymbol{x}$ по искаженному $\tilde{\boldsymbol{x}}$ или, в более общем случае, вернуть условное распределение вероятности $p(\boldsymbol{x} \mid \tilde{\boldsymbol{x}})$.
- **Оценка функции вероятности или функции плотности вероятности.** В задаче оценки плотности алгоритм должен обучить функцию $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$, где $p_{\text{model}}(\boldsymbol{x})$ интерпретируется как функция плотности вероятности (если $\boldsymbol{x}$ – непрерывная случайная величина) или как функция вероятности (в дискретном случае) в пространстве, из которого были взяты примеры. Чтобы решить эту задачу хорошо (что это значит, мы точно определим в разделе о мерах качества), алгоритм должен выявить структуру предъявленных данных. Он должен понять, где примеры расположены тесно, а где их появление маловероятно. Для большинства описанных выше задач требуется, чтобы алгоритм хотя бы неявно уловил структуру распределения вероятности. В задаче оценки плотности это распределение должно быть определено явно. В принципе, найденное распределение впоследствии можно использовать для решения других задач. Например, получив распределение вероятности $p(\boldsymbol{x})$, мы можем воспользоваться им для подстановки отсутствующих значений. Если значение x_i отсутствует, а все остальные значения, множество которых обозначается $\boldsymbol{x}_{-i}$, имеются, то мы знаем, что распределение x_i имеет вид $p(x_i \mid \boldsymbol{x}_{-i})$. На практике оценка плотности не всегда позволяет решить все сопутствующие задачи, потому что операции, которые нужно выполнить с $p(\boldsymbol{x})$, требуют непомерно большого объема вычислений.

Разумеется, есть много других типов задач. Перечисленные выше типы – лишь пример того, на что способно машинное обучение, а не строгая систематизация.

5.1.2. Мера качества *P*

Чтобы оценить возможности алгоритма машинного обучения, мы должны иметь количественную меру его качества. Обычно мера качества P специфична для решаемой системой задачи T.

Для таких задач, как классификация, классификация при отсутствии части данных и транскрипция, часто измеряется **верность** (accuracy) модели. Верностью называется доля примеров, для которых модель выдала правильный результат. Эквивалентную информацию можно получить путем измерения **частоты ошибок** – доли примеров, для которых модель выдала неправильный результат. Иногда частота ошибок описывается бинарной функцией потерь, которая принимает для данного примера значение 0, если он классифицирован правильно, и 1, если неправильно. В задаче оценки плотности не имеет смысла измерять верность, частоту ошибок или любой другой бинарный показатель. Нужна какая-то другая мера качества, принимающая значения из непрерывного диапазона. Чаще всего для этой цели используется средняя логарифмическая вероятность, присваиваемая примерам моделью.

Обычно нас интересует, насколько хорошо алгоритм машинного обучения работает на данных, которых прежде не видел, поскольку это показывает, как будет вести себя модель, развернутая в реальном приложении. Поэтому меры качества вычисляются для тестового набора данных, отдельного от данных, на которых система обучалась.

Выбор меры качества может показаться простым и объективным делом, но зачастую очень трудно найти такую меру, которая точно соответствует желаемому поведению системы.

Иногда причина в том, что трудно решить, что же нужно измерять. Например, в задаче транскрипции нужно измерять верность при транскрипции всей последовательности или лучше использовать более детальную меру, которая поощряет за правильную транскрипцию отдельных элементов последовательности? В задаче регрессии за что штрафовать систему сильнее: за частые ошибки средней серьезности или за редкие, но очень серьезные ошибки? Ответ на такие вопросы зависит от приложения.

Но бывает и так, что мы точно знаем, что хотели бы измерить в идеале, только сам процесс измерения практически не реализуем. Такое часто случается в контексте оценивания плотности. Многие из лучших вероятностных моделей представляют распределение вероятности лишь неявно. Вычислить фактическую вероятность конкретной точки в пространстве в подобных моделях технически невозможно. В таких случаях нужно придумать альтернативный критерий, который все же соответствует проектным целям, или предложить хорошую аппроксимацию желаемого критерия.

5.1.3. Опыт *E*

Алгоритмы машинного обучения можно разделить на два больших класса, **без учителя** и **с учителем**, в зависимости от того, на каком опыте они могут обучаться.

Можно считать, что большинству алгоритмов обучения, описываемых в этой книге, в качестве опыта доступен весь **набор данных**. Набором данных называется совокупность большого числа примеров в смысле раздела 5.1.1. Иногда примеры называются **замерами**, или **точками**.

Один из самых старых наборов данных, используемых исследователями в области статистики и машинного обучения, – Iris (Fisher, 1936). Это результаты измерений различных частей 150 растений семейства ирисовых. Каждому растению соответствует один пример. Признаками являются обмеры частей растения: длина чашелистика, ширина чашелистика, длина лепестка и ширина лепестка. Указано также, к какому виду относится растение. Всего в наборе данных представлены три вида.

Алгоритму обучения без учителя в качестве опыта предъявляется набор данных, содержащий много признаков, а алгоритм должен выявить полезные структурные свойства этого набора. В глубоком обучении нас обычно интересует полное распределение вероятности, описывающее предъявленный набор – явно, как в задаче оценивания плотности, или неявно, как в задачах синтеза или очистки от шума. Некоторые алгоритмы обучения без учителя решают другие задачи, например алгоритм кластеризации, который должен выделить в наборе данных кластеры похожих примеров.

Алгоритму обучения с учителем предъявляется набор данных, содержащий признаки, в котором каждый пример снабжен **меткой**, или **целевым классом**. Так, в наборе данных Iris примеры аннотированы видами растений. Алгоритм должен проанализировать набор Iris и научиться определять класс ириса по результатам измерений.

Грубо говоря, обучение без учителя означает наблюдение нескольких примеров случайного вектора $\mathbf{x}$ с последующей попыткой вывести, явно или неявно, распределение вероятности $p(\mathbf{x})$ или некоторые интересные свойства этого распределения. А обучение с учителем сводится к наблюдению нескольких примеров случайного вектора $\mathbf{x}$ и ассоциированного с ним значения или вектора $\mathbf{y}$ с последующей попыткой предсказать $\mathbf{y}$ по $\mathbf{x}$, обычно в виде оценки $p(\mathbf{y} \mid \mathbf{x})$. Мы считаем, что метка $\mathbf{y}$ предо-

ставлена неким учителем, который объясняет системе машинного обучения, что делать, – отсюда и название «**обучение с учителем**». В обучении без учителя никакого учителя не существует, и алгоритм должен извлечь из данных смысл без подсказки.

Обучение с учителем и без учителя – термины, не имеющие формального определения. Границы между тем и другим часто размыты. Многие технологии машинного обучения применимы к решениям задач обоих типов. Например, цепное правило вероятностей утверждает, что для вектора $\mathbf{x} \in \mathbb{R}^n$ совместное распределение можно представить в виде произведения:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \ldots, \mathrm{x}_{i-1}). \tag{5.1}$$

Такое разложение означает, что моделирование $p(\mathbf{x})$, которое на первый взгляд кажется задачей без учителя, можно разделить на n задач обучения с учителем. С другой стороны, задачу обучения с учителем, состоящую в вычислении $p(y \mid \mathbf{x})$, можно решить с помощью традиционных методов обучения без учителя: найти совместное распределение $p(\mathbf{x}, y)$, а затем вычислить

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}. \tag{5.2}$$

Хотя обучение с учителем и без учителя – понятия, не формализованные и не разделенные четкой границей, они все же помогают приблизительно распределить по категориям некоторые задачи, решаемые в машинном обучении. Принято относить задачи регрессии, классификации и структурного вывода к обучению с учителем, а задачу оценивания плотности – к обучению без учителя.

Возможны и другие варианты парадигмы обучения. Например, в случае обучения с частичным привлечением учителя одни примеры снабжены метками, а другие – нет. В многовариантном обучении вся совокупность примеров помечается как содержащая или не содержащая пример класса, но отдельные ее элементы никак не помечаются. Недавний пример многовариантного обучения глубоких моделей см. в работе Kotzias et al. (2015).

В некоторых алгоритмах машинного обучения в роли опыта выступает не только фиксированный набор данных. Так, алгоритмы **обучения с подкреплением** взаимодействуют с окружающей средой, так что между системой обучения и ее опытом образуется контур с обратной связью. Такие алгоритмы выходят за рамки нашей книги. Дополнительные сведения об обучении с подкреплением см. в работах Sutton and Barto (1998) или Bertsekas and Tsitsiklis (1996), а о подходе к этой теме на основе глубокого обучения – Mnih et al. (2013).

В большинстве алгоритмов машинного обучения опытом является просто набор данных, который можно описать разными способами. Но в любом случае набор данных – это совокупность примеров, каждый из которых является совокупностью признаков.

Один распространенный способ описания набора данных – **матрица плана**. В строках этой матрицы расположены примеры, а в столбцах – признаки. Например, в наборе данных Iris 150 примеров с четырьмя признаками на каждый. Следовательно, его можно представить матрицей плана $\boldsymbol{X} \in \mathbb{R}^{150 \times 4}$, где $X_{i,1}$ – длина чашелистика i-го растения, $X_{i,1}$ – ширина чашелистика i-го растения и т. д. Большинство алгоритмов обуче-

ния в этой книге описывается в терминах работы с наборами данных, представленными в виде матрицы плана.

Разумеется, описать набор данных матрицей плана можно, только если каждый пример описывается вектором, причем все эти векторы должны быть одинакового размера. Так бывает не всегда. Например, в коллекции фотографий разной ширины и высоты фотографии состоят из разного числа пикселей и, значит, описываются векторами разной длины. В разделе 9.7 и в главе 10 мы обсудим, как работать с гетерогенными данными. В подобных случаях набор данных описывается не матрицей с m строками, а множеством m элементов: $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)},..., \boldsymbol{x}^{(m)}\}$. Такая нотация не предполагает, что любые два вектора-примера $\boldsymbol{x}^{(i)}$ и $\boldsymbol{x}^{(j)}$ имеют одинаковый размер.

В случае обучения с учителем пример содержит не только набор признаков, но и метку. Так, если мы хотим обучить алгоритм распознавать объекты на фотографиях, то должны будем указать, какие объекты присутствуют на каждой фотографии. Для этого можно задавать числовой код: 0 – человек, 1 – автомобиль, 2 – кошка и т. д. Часто при работе с набором данных в виде матрицы плана с наблюдаемыми признаками $\boldsymbol{X}$ мы предоставляем также вектор меток $\boldsymbol{y}$, в котором элемент y_i содержит метку i-го примера.

Конечно, не всегда метка – единственное число. Так, если мы хотим научить систему распознавания речи транскрибировать целые предложения, то меткой для каждого примера будет последовательность слов. Как не существует формального определения обучения с учителем и без учителя, так нет и строгой систематизации наборов данных или видов опыта. Описанные выше структуры покрывают большинство случаев, но для новых приложений всегда можно придумать что-то необычное.

5.1.4. Пример: линейная регрессия

Наше определение алгоритма машинного обучения как алгоритма, способного улучшить качество работы программы для решения некоторой задачи на основе опыта, звучит несколько абстрактно. Чтобы добавить конкретики, приведем пример простого алгоритма машинного обучения: линейной регрессии. Мы будем не раз возвращаться к нему по ходу знакомства с дополнительными концепциями машинного обучения, чтобы лучше понять поведение алгоритма.

Как следует из названия, линейная регрессия решает задачу регрессии. Иными словами, наша цель – построить систему, которая принимает на входе вектор $\boldsymbol{x} \in \mathbb{R}^n$, а на выходе порождает скалярное значение $y \in \mathbb{R}$. Результатом линейной регрессии является линейная функция входных данных. Обозначим $\hat{y}$ – значение y, предсказанное моделью. Определим результат модели в виде

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}, \tag{5.3}$$

где $\boldsymbol{w} \in \mathbb{R}^n$ – вектор **параметров**.

Параметры – это величины, управляющие поведением системы. В данном случае w_i – коэффициент, на который нужно умножить признак x_i перед включением в сумму вкладов всех признаков. Иными словами, $\boldsymbol{w}$ – набор весов, описывающих влияние отдельных признаков на результат предсказания. Если вес w_i признака x_i положителен, то увеличение признака приводит к увеличению результата $\hat{y}$, а если отрицателен, то к уменьшению. Если абсолютная величина веса признака велика, то его влияние на предсказание значительно. Если же вес признака равен 0, то признак вообще не влияет на предсказание.

Таким образом, определение нашей задачи T выглядит следующим образом: предсказать y по $\boldsymbol{x}$, вычислив значение $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$. Далее необходимо определение меры качества P.

Пусть имеется матрица плана с m примерами, которые мы будем использовать не для обучения, а только для оценки качества работы модели. Имеется также вектор меток, содержащий правильные значения y для каждого из этих примеров. Поскольку этот набор данных будет использоваться только для контроля качества, назовем его тестовым набором. Обозначим матрицу плана $\boldsymbol{X}^{(\text{test})}$, а вектор меток регрессии – $\boldsymbol{y}^{(\text{test})}$.

Один из способов измерения качества модели – вычислить **среднеквадратическую ошибку** модели на тестовом наборе. Если вектор $\hat{\boldsymbol{y}}^{(\text{test})}$ содержит предсказания модели на тестовом наборе, то среднеквадратическая ошибка определяется по формуле:

$$\text{MSE}_{\text{test}} = \frac{1}{m}\sum_i (\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})})_i^2. \tag{5.4}$$

Эта мера ошибки обращается в 0, когда $\hat{\boldsymbol{y}}^{(\text{test})} = \boldsymbol{y}^{(\text{test})}$. Кроме того,

$$\text{MSE}_{\text{test}} = \frac{1}{m}\left\|\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})}\right\|_2^2, \tag{5.5}$$

поэтому ошибка тем больше, чем больше евклидово расстояние между предсказаниями и метками.

Мы должны спроектировать алгоритм машинного обучения, который улучшает веса $\boldsymbol{w}$ таким образом, что MSE_{test} уменьшается по мере того, как алгоритм получает новый опыт, наблюдая обучающий набор $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$. Интуитивно понятный способ добиться этой цели (мы обоснуем его в разделе 5.5.1) – минимизировать среднеквадратическую ошибку на обучающем наборе, $\text{MSE}_{\text{train}}$.

Для минимизации $\text{MSE}_{\text{train}}$ нужно просто приравнять градиент 0 и решить получившееся уравнение:

$$\nabla_{\boldsymbol{w}} \text{MSE}_{\text{train}} = 0 \tag{5.6}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \frac{1}{m}\left\|\hat{\boldsymbol{y}}^{(\text{train})} - \boldsymbol{y}^{(\text{train})}\right\|_2^2 = 0 \tag{5.7}$$

$$\Rightarrow \frac{1}{m}\nabla_{\boldsymbol{w}} \left\|\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}\right\|_2^2 = 0 \tag{5.8}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} (\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})})^\top (\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}) = 0 \tag{5.9}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} (\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - 2\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} + \boldsymbol{y}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})}) = 0 \tag{5.10}$$

$$\Rightarrow 2\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - 2\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} = 0 \tag{5.11}$$

$$\Rightarrow \boldsymbol{w} = (\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})})^{-1}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} \tag{5.12}$$

Система уравнений, решение которой дает формула (5.12), называется **нормальными уравнениями**. Вычисление выражения (5.12) и есть простой алгоритм обучения. На рис. 5.1 показан алгоритм обучения линейной регрессии в действии.

Отметим, что термином **линейная регрессия** часто обозначают несколько более сложную модель с одним дополнительным параметром – свободным членом b. В этой модели

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b, \tag{5.13}$$

поэтому отображение параметров на предсказания по-прежнему описывается линейной функцией, но отображение признаков на предсказания теперь является аффинной функцией. Это обобщение означает, что график предсказанной модели – прямая, которая не обязана проходить через начало координат. Вместо добавления параметра b можно продолжать пользоваться моделью одних лишь весов, но дополнить вектор $\boldsymbol{x}$ элементом, который всегда равен 1. Вес, соответствующий этому элементу, играет роль свободного члена. В этой книге мы часто будем употреблять термин «линейный» в применении к аффинным функциям.

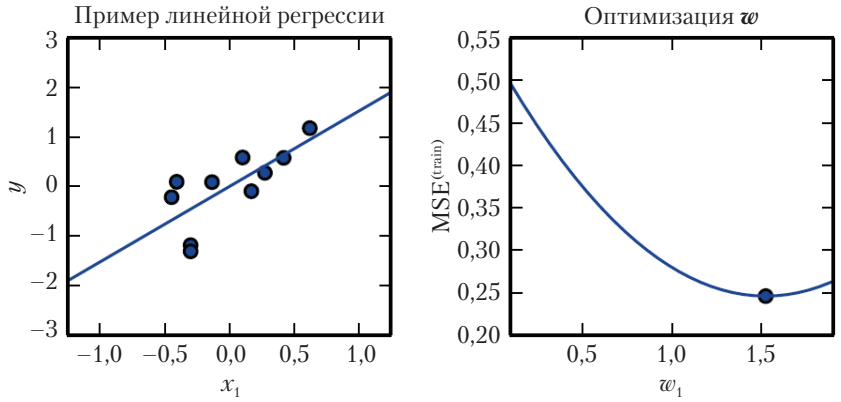

Рис. 5.1 ❖ Задача линейной регрессии, в которой обучающий набор состоит из десяти примеров, по одному признаку в каждом. Поскольку признак всего один, вектор весов $\boldsymbol{w}$ содержит единственный подлежащий обучению параметр w_1. (*Слева*) Модель линейной регрессии обучила вес w_1, так что прямая $y = w_1x$ проходит максимально близко ко всем обучающим примерам. (*Справа*) Точка на графике соответствует значению w_1, найденному из нормальных уравнений, которое, как мы видим, минимизирует среднеквадратическую ошибку на обучающем наборе

Свободный член b часто называют параметром **смещения** (bias) аффинного преобразования. Это связано с тем, что результат преобразования смещается в сторону b, если входных данных нет вообще. Это словоупотребление не имеет ничего общего со статистическим смещением, когда оценка некоторой величины, полученная алгоритмом статистического оценивания, не совпадает с истинным значением.

Конечно, линейная регрессия – очень простой и ограниченный алгоритм обучения, но он дает представление о том, как такие алгоритмы могут работать. В последующих разделах мы опишем базовые принципы, лежащие в основе проектирования алгоритмов обучения, и продемонстрируем, как эти принципы применяются к построению более сложных алгоритмов.

5.2. Емкость, переобучение и недообучение

Главная проблема машинного обучения состоит в том, что алгоритм должен хорошо работать на новых данных, которых он раньше не видел, а не только на тех, что использовались для обучения модели. Эта способность правильной работы на ранее не предъявлявшихся данных называется **обобщением**.

Обычно при обучении модели мы имеем доступ к обучающему набору: можем вычислить некоторую меру ошибки на обучающем наборе, которая называется **ошибкой обучения**, и постараться минимизировать ее. То, что мы только что описали, – всего лишь задача оптимизации. Машинное обучение отличается от оптимизации тем, что мы хотим еще и уменьшить **ошибку обобщения** (ее также называют **ошибкой тестирования**). Ошибка обобщения – это математическое ожидание ошибки на новых входных данных. Здесь математическое ожидание вычисляется по возможным входным данным, выбираемым из распределения, которое, как мы думаем, может встретиться на практике.

Как правило, для оценки ошибки обобщения модели измеряется ее качество на тестовом наборе данных, отдельном от обучающего набора.

В примере линейной регрессии мы обучали модель путем минимизации следующей ошибки обучения:

$$\frac{1}{m^{(\text{train})}} \left\| \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right\|_2^2, \tag{5.14}$$

но в действительности нас интересует ошибка тестирования $\frac{1}{m^{(\text{test})}} \left\| \boldsymbol{X}^{(\text{test})} \boldsymbol{w} - \boldsymbol{y}^{(\text{test})} \right\|_2^2$.

Как можно повлиять на качество работы на тестовом наборе, если для наблюдения доступен только обучающий набор? Некоторые ответы дает **теория статистического обучения**. Если обучающий и тестовый наборы готовились произвольным образом, то действительно мало что можно сделать. Но если разрешено делать предположения о порядке сбора данных для обучающего и тестового наборов, то удается продвинуться дальше.

Обучающий и тестовый наборы генерируются из распределения вероятности тестовых наборов **процессом генерации данных**. Обычно считаются справедливыми предположения, в совокупности называемые **i.i.d.** (independent and identically-distributed), а именно: примеры в каждом наборе **независимы** и оба набора **одинаково распределены**, т. е. выбираются из одного и того же распределения вероятности. Это общее распределение называется **порождающим распределением** и обозначается p_{data}. При таких предположениях мы можем математически проанализировать связь между ошибкой обучения и ошибкой тестирования.

Сразу заметим, что ожидаемая ошибка обучения случайно выбранной модели равна ожидаемой ошибке тестирования той же модели. Пусть имеется распределение вероятности $p(\boldsymbol{x}, y)$, и мы повторно производим из него выборку для генерации обучающего и тестового наборов. Для фиксированного значения $\boldsymbol{w}$ ожидаемая ошибка на обучающем наборе в точности равна ожидаемой ошибке на тестовом наборе, потому что оба математических ожидания являются результатом одного и того же процесса выборки. Разница лишь в названии набора данных.

Конечно, при использовании алгоритма машинного обучения мы не фиксируем параметров заранее, с тем чтобы потом произвести выборку обоих наборов данных. Мы выбираем обучающий набор, используем его для минимизации ошибки обучения, а затем выбираем тестовый набор. При таком процессе ожидаемая ошибка тестирования больше или равна ожидаемой ошибке обучения. Факторов, определяющих качество работы алгоритма машинного обучения, два:

1) сделать ошибку обучения как можно меньше;
2) сократить разрыв между ошибками обучения и тестирования.

Эти факторы соответствуют двум центральным проблемам машинного обучения: **недообучению** и **переобучению**. Недообучение имеет место, когда модель не позволяет получить достаточно малую ошибку на обучающем наборе, а переобучение – когда разрыв между ошибками обучения и тестирования слишком велик.

Управлять склонностью модели к переобучению или недообучению позволяет ее **емкость** (capacity). Неформально говоря, емкость модели описывает ее способность к аппроксимации широкого спектра функций. Модели малой емкости испытывают сложности в аппроксимации обучающего набора. Модели большой емкости склонны к переобучению, поскольку запоминают свойства обучающего набора, не присущие тестовому.

Один из способов контроля над емкостью алгоритма обучения состоит в том, чтобы выбрать его **пространство гипотез** – множество функций, которые алгоритм может рассматривать в качестве потенциального решения. Например, пространством гипотез алгоритма линейной регрессии является множество всех линейных функций от входных данных. Мы можем обобщить линейную регрессию, включив в пространство гипотез многочлены более высокой степени. При этом увеличится емкость модели.

Ограничившись только многочленами степени 1, мы получим модель линейной регрессии, с которой уже знакомы:

$$\hat{y} = b + wx. \tag{5.15}$$

Добавив еще один признак, коэффициент при x^2, мы сможем обучить модель в виде квадратичной функции от x:

$$\hat{y} = b + w_1 x + w_2 x^2. \tag{5.16}$$

Хотя эта модель ищет квадратичную функцию *входных данных*, результат попрежнему линейно зависит от *параметров*, так что мы можем использовать нормальные уравнения для обучения модели в замкнутой форме. Продолжая добавлять в качестве признаков коэффициенты при более высоких степенях, мы можем получить, например, многочлен степени 9:

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i. \tag{5.15}$$

В общем случае алгоритмы машинного обучения работают оптимально, когда емкость модели соответствует истинной сложности задачи и объему обучающих данных. Модель недостаточной емкости не способна решать сложные задачи. Модель избыточной емкости может решать сложные задачи, но если емкость слишком высока для конкретной задачи, то возникает риск переобучения.

На рис. 5.2 иллюстрируется этот принцип. Мы применяем три модели – линейную, квадратичную и полиномиальную степени 9 – к задаче аппроксимации, когда истинная функция – квадратичная. Линейная модель не способна уловить кривизну истинной кривой, поэтому является недообученной. Модель степени 9 может представить правильную функцию, но вместе с ней еще бесконечно много функций, проходящих через те же точки, поскольку параметров больше, чем обучающих примеров. Мало шансов, что из такого несметного множества совершенно непохожих кандидатов будет выбрано хорошо обобщающееся решение. В данном случае квадратичная модель точно соответствует истинной структуре задачи, поэтому она хорошо обобщается на новые данные

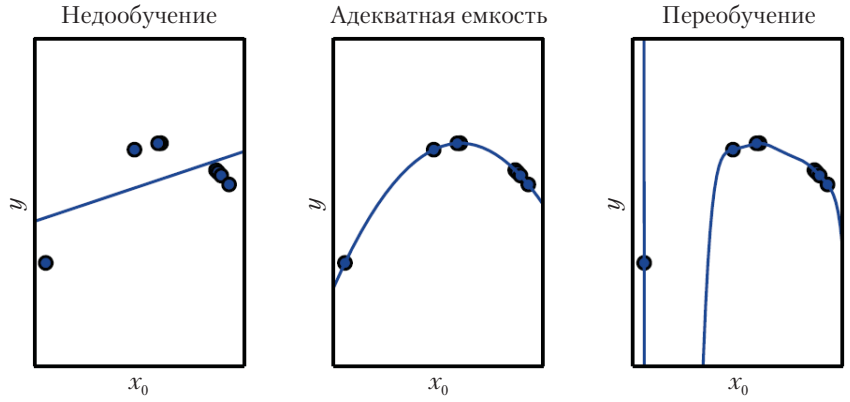

Рис. 5.2 ❖ На одном обучающем наборе обучены три модели. Обучающие данные сгенерированы синтетически: значения x выбирались случайно, а значения y вычислялись детерминированно путем применения квадратичной функции. (*Слева*) Линейная функция, аппроксимирующая данные, страдает от недообучения – она не улавливает присущую данным кривизну. (*В центре*) Квадратичная функция, аппроксимирующая данные, хорошо обобщается на новые точки. Ей не свойственны ни недообучение, ни переобучение. (*Справа*) Многочлен степени 9 аппроксимирует данные, но страдает от переобучения. Для решения недоопределенной системы нормальных уравнений мы воспользовались псевдообратной матрицей Мура–Пенроуза. Найденная кривая проходит через все обучающие точки, но мы не смогли выявить правильную структуру. Между двумя обучающими точками присутствует глубокая впадина, которой нет в истинной функции. Кроме того, функция резко возрастает слева от данных, тогда как истинная функция в этой области убывает

Пока что мы описали только один способ изменения емкости модели: модификация числа признаков с одновременным добавлением ассоциированных с этими признаками параметров. Но на самом деле есть много способов настройки емкости. Емкость определяется не только выбором модели. Модель задает семейство функций, из которого может выбирать алгоритм обучения в процессе варьирования параметров. Это называется **репрезентативной емкостью** модели. Во многих случаях отыскание наилучшей функции в семействе является трудной задачей оптимизации. На практике алгоритм обучения находит не лучшую функцию, а лишь такую, которая существенно уменьшает ошибку обучения. Наличие дополнительных ограничений, в частности несовершенство алгоритма оптимизации, означает, что **эффективная емкость** алгоритма обучения может быть меньше репрезентативной емкости модели.

Современные идеи об обобщаемости моделей машинного обучения восходят еще к античным философам, в частности Птолемею. Многие ученые прежних времен исповедовали принцип экономии, который сейчас больше известен под названием «**бритва Оккама**» (приблизительно 1287–1347). Этот принцип утверждает, что из всех гипотез, одинаково хорошо объясняющих наблюдения, следует выбирать «простейшую». Эта идея была формализована и уточнена в XX веке основателями теории статистического обучения (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer et al., 1989; Vapnik, 1995).

Теория статистического обучения предлагает различные способы количественного выражения емкости модели. Самый известный из них – **размерность Вапника–Червоненкиса**, или VC-размерность, измеряющая емкость бинарного классификатора. VC-размерность определяется как наибольшее возможное значение m – такое, что существует обучающий набор m разных точек $\boldsymbol{x}$, которые классификатор может пометить произвольным образом.

Количественное выражение емкости модели позволяет теории статистического обучения делает количественные предсказания. Самые важные результаты этой теории показывают, что расхождение между ошибкой обучения и ошибкой обобщения ограничено сверху величиной, которая растет с ростом емкости модели, но убывает по мере увеличения количества обучающих примеров (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer et al., 1989; Vapnik, 1995). Наличие такого ограничения является теоретическим обоснованием работоспособности алгоритмов машинного обучения, но на практике используется редко в применении к алгоритмам глубокого обучения. Отчасти это связано с нестрогостью оценки границ, а отчасти со сложностью определения емкости алгоритмов глубокого обучения. Последняя проблема особенно трудна, потому что эффективная емкость ограничена возможностями алгоритма оптимизации, а у нас мало теоретических результатов об общих задачах невыпуклой оптимизации, встречающихся в глубоком обучении.

Следует помнить, что хотя лучшая способность к обобщению (малым разрывом между ошибками обучения и тестирования) свойственна скорее простым функциям, нам все равно приходится выбирать достаточно сложные гипотезы для достижения малой ошибки обучения. В типичном случае ошибка обучения убывает, асимптотически приближаясь к минимально возможной ошибке с ростом емкости модели (в предположении, что у меры ошибки есть минимальное значение). А типичная ошибка обобщения имеет форму U-образной кривой, как показано на рис. 5.3.

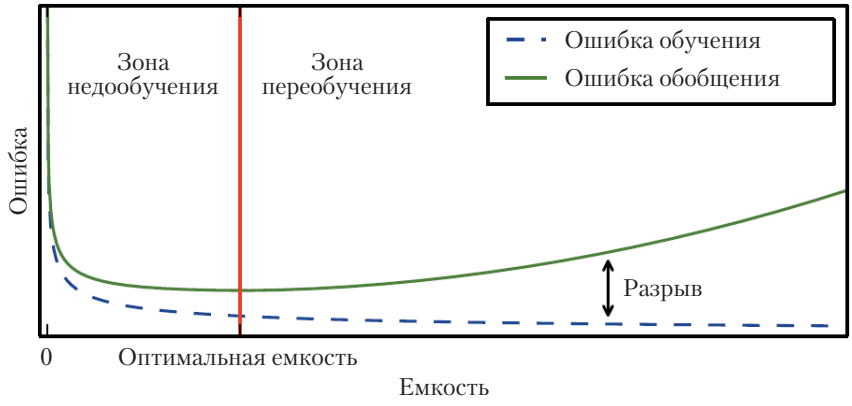

Рис. 5.3 ❖ Типичная связь между емкостью и ошибкой. Ошибки обучения и тестирования ведут себя по-разному. В левой части графика обе ошибки принимают большие значения. Это режим недообучения. По мере увеличения емкости ошибка обучения снижается, а разрыв между ошибкой обучения и обобщения растет. В конечном итоге величина разрыва перевешивает уменьшение ошибки обучения, и мы попадаем в режим переобучения, где емкость слишком сильно превышает оптимальную емкость

Чтобы подойти к крайнему случаю произвольно высокой емкости, введем понятие **непараметрической** *модели*. До сих пор мы рассматривали только параметрические модели типа линейной регрессии. Параметрическая модель обучает функцию, описанную вектором параметров, размер которого конечен и фиксируется до наблюдения данных. У непараметрических моделей такого ограничения нет.

Иногда непараметрические модели представляют собой просто теоретические абстракции (например, алгоритм, который производит поиск среди всех возможных распределений вероятности), не реализуемые на практике. Однако можно также спроектировать практически полезные непараметрические модели, сделав их сложность функцией от размера обучающего набора. Примером такого алгоритма может служить **регрессия методом ближайшего соседа**. В отличие от линейной регрессии, когда размер вектора весов фиксирован, модель регрессии методом ближайшего соседа просто сохраняет $\boldsymbol{X}$ и $\boldsymbol{y}$ из обучающего набора. Когда модель просят классифицировать тестовую точку $\boldsymbol{x}$, она находит ближайшую к $\boldsymbol{x}$ точку обучающего набора и возвращает ассоциированную с ней метку. Иными словами, $\hat{y} = y_i$, где $i = \arg\min \|\boldsymbol{X}_{i,:} - \boldsymbol{x}\|_2^2$. Этот алгоритм легко обобщается на метрики, отличные от нормы L^2, например найденные в процессе обучения (Goldberger et al., 2005). Если позволено разрешать неоднозначности путем усреднения значений y_i по всем $\boldsymbol{X}_{i,:}$ являющимся ближайшими соседями, то этот алгоритм может достичь минимально возможной ошибки обучения (которая может оказаться больше нуля, если с двумя одинаковыми примерами ассоциированы разные метки) на любом наборе данных для регрессии.

Наконец, мы можем создать непараметрический алгоритм обучения, обернув параметрический алгоритм другим, который увеличивает число параметров по мере необходимости. Представьте, к примеру, внешний цикл обучения, который изменяет степень многочлена, обучаемого в результате регрессии.

Идеальной моделью являлся бы оракул, который просто знает истинное распределение вероятности, на основе которого генерируются данные. Но даже такая модель будет давать некоторую ошибку для многих задач из-за шума. В случае обучения с учителем отображение $\boldsymbol{x}$ в y может быть действительно стохастическим, но также возможно, что y – детерминированная функция, только у нее есть дополнительные аргументы, помимо включенных в $\boldsymbol{x}$. Расхождение предсказаний оракула с истинным распределением $p(\boldsymbol{x}, y)$ называется **байесовской ошибкой**.

Ошибки обучения и обобщения могут варьироваться в зависимости от размера обучающего набора. Ожидаемая ошибка обобщения никогда не может увеличиться с ростом количества обучающих примеров. Для непараметрических моделей увеличение объема данных приводит к лучшему обобщению до тех пор, пока не будет достигнута наименьшая возможная ошибка. Любая фиксированная параметрическая модель емкостью ниже оптимальной асимптотически приближается к значению ошибки, большему байесовской. Это показано на рис. 5.4. Может случиться и так, что емкость модели оптимальна, и тем не менее существует большой разрыв между ошибками обучения и обобщения. В такой ситуации разрыв, возможно, удастся сократить, увеличив число обучающих примеров.

5.2.1. Теорема об отсутствии бесплатных завтраков

Теория обучения утверждает, что алгоритм может хорошо обобщаться после обучения на конечном множестве примеров. На первый взгляд, это противоречит базовым принципам логики. Индуктивное рассуждение, когда общие правила выводятся из

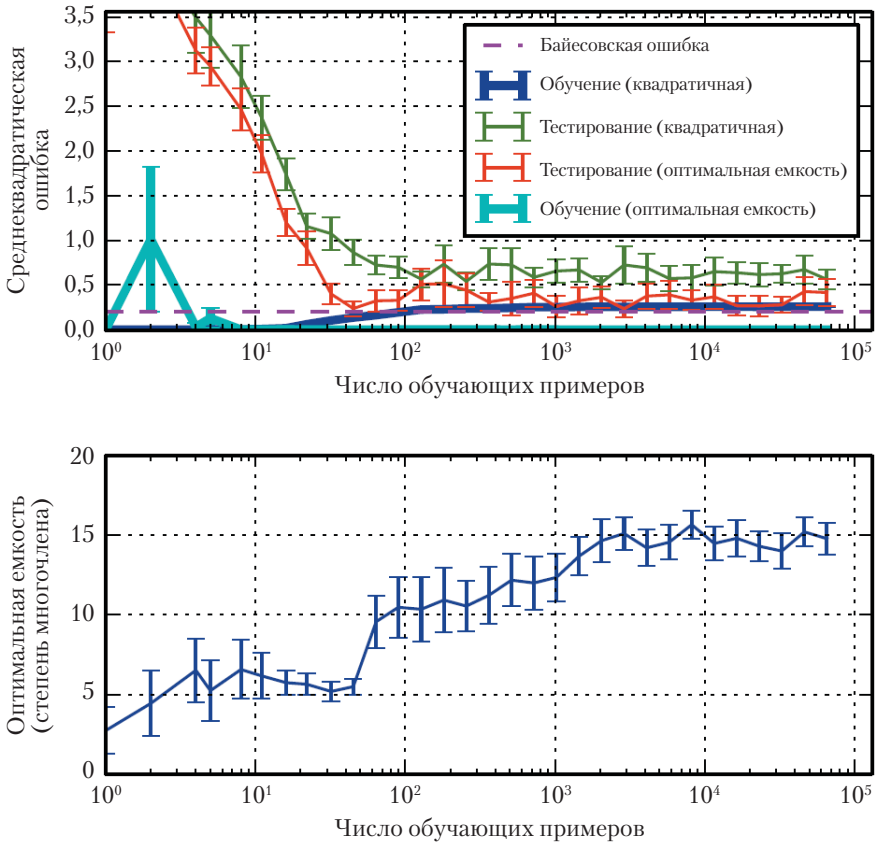

Рис. 5.4 ❖ Влияние размера обучающего набора данных на ошибки обучения и тестирования, а также на оптимальную емкость модели. Мы синтезировали задачу регрессии, добавив умеренный шум к многочлену степени 5, сгенерировали один тестовый набор, а затем несколько обучающих наборов разного размера. Для каждого размера было сгенерировано 40 различных обучающих наборов, чтобы нанести на график отрезки, отражающие 95%-ные доверительные интервалы. (*Вверху*) Среднеквадратическая ошибка на обучающем и тестовом наборах для двух разных моделей: квадратичной и полиномиальной, для которой выбрана степень, минимизирующая тестовую ошибку. Обе модели выражены в замкнутой форме. Для квадратичной модели ошибка обучения возрастает с ростом обучающего набора, поскольку чем больше набор, тем труднее его аппроксимировать. Одновременно ошибка тестирования убывает, поскольку меньше неправильных гипотез совместимо с обучающими данными. Емкость квадратичной модели недостаточна для решения этой задачи, поэтому ошибка тестирования асимптотически приближается к высокому значению. Ошибка тестирования при оптимальной емкости асимптотически приближается к байесовской ошибке. Ошибка обучения может стать ниже байесовской, поскольку алгоритм обучения способен запоминать конкретные экземпляры обучающего набора. Когда размер обучающего набора стремится к бесконечности, ошибка обучения любой модели фиксированной емкости (в данном случае квадратичной) должна возрастать как минимум до байесовской ошибки. (*Внизу*) С ростом размера обучающего набора оптимальная емкость (показанная здесь как степень оптимального полиномиального регрессора) увеличивается. Оптимальная емкость выходит на плато после достижения сложности, достаточной для решения задачи

ограниченного множества примеров, не является логически корректным. Чтобы вывести правило, описывающее каждый элемент множества, необходимо иметь информацию о каждом элементе.

Машинное обучение позволяет частично обойти эту проблему, поскольку предлагает только вероятностные правила, а не однозначно определенные, как в чистой логике. Машинное обучение обещает найти правила, с некоторой вероятностью справедливые для большинства элементов множества.

К сожалению, даже это не решает проблему полностью. **Теорема об отсутствии бесплатных завтраков** в машинном обучении (Wolpert, 1996) утверждает, что в среднем по всем возможным порождающим определениям у любого алгоритма классификации частота ошибок классификации ранее не наблюдавшихся примеров одинакова. Самый изощренный алгоритм, который мы только можем придумать, в среднем (по всем возможным задачам) дает такое же качество, как простейшее предсказание: все точки принадлежат одному классу.

По счастью, эти результаты справедливы, только если производить усреднение по всем возможным порождающим распределениям. Если сделать предположения о видах распределений вероятности, встречающихся в реальных приложениях, то можно спроектировать алгоритмы обучения, хорошо работающие для таких распределений.

Это означает, что цель работ по машинному обучению – не в том, чтобы найти универсальный или самый лучший алгоритм обучения, а в том, чтобы понять, какие виды распределений характерны для «реального мира», в котором функционирует агент ИИ, и какие виды алгоритмов машинного обучения хорошо работают для интересных нам порождающих распределений.

5.2.2. Регуляризация

Из теоремы об отсутствии бесплатных завтраков следует, что алгоритмы машинного обучения нужно проектировать, так чтобы они хорошо работали для конкретной задачи. Для этого мы встраиваем в алгоритм набор предпочтений. Если эти предпочтения согласованы с задачами, которые решает алгоритм, то он будет работать лучше.

До сих пор мы обсуждали только один способ модификации алгоритма обучения: увеличение или уменьшение репрезентативной емкости модели путем добавления или удаления функций в пространстве гипотез, описывающем, какие решения может выбирать алгоритм. Мы привели пример: увеличение или уменьшение степени многочлена в задаче регрессии. Но это чрезмерно упрощенный взгляд на вещи.

На поведение алгоритма сильно влияет не только разнообразие множества функций в пространстве гипотез, но и конкретный вид этих функций. Для алгоритма линейной регрессии пространство гипотез состояло из множества линейных функций от входных данных. Они могут быть полезны для задач, в которых связь между входом и выходом действительно близка к линейной. Но если поведение нелинейно, то полезность оказывается куда меньше. Например, линейная регрессия покажет плачевные результаты при попытке предсказать поведение функции $\sin(x)$. Таким образом, мы можем контролировать качество алгоритма, выбирая вид функций, которые могут выступать в роли решения, а также управляя количеством таких функций.

Кроме того, мы можем настроить алгоритм обучения, так чтобы он отдавал предпочтение определенным решениям из пространства гипотез. Это означает, что до-

пустимы функции любого вида, но при прочих равных условиях выбираться будет функция предпочтительного вида. Другое решение будет выбрано, только если оно аппроксимирует обучающие данные значительно лучше предпочтительного.

Например, в критерий обучения для линейной регрессии можно включить снижение весов. Для выполнения линейной регрессии со снижением весов мы пытаемся минимизировать сумму, состоящую из среднеквадратической ошибки на обучающих данных и критерия $J(\boldsymbol{w})$, который отдает предпочтение весам с меньшим квадратом нормы L^2. Точнее,

$$J(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w}, \tag{5.18}$$

где λ – заранее выбранное значение, задающее силу предпочтения малых весов. При $\lambda = 0$ никаких предпочтений нет, а чем больше λ, тем сильнее стремление к малым весам. В процессе минимизации $J(\boldsymbol{w})$ выбираются веса, выражающие компромисс между аппроксимацией обучающих данных и малостью весов. Это дает решение, для которого либо угловой коэффициент меньше, либо веса назначаются меньшему числу признаков. В качестве примера того, как можно управлять тенденцией модели к переобучению или недообучению с помощью снижения весов, мы обучили полиномиальную модель регрессии высокой степени с разными значениями λ. Результат показан на рис. 5.5.

Рис. 5.5 ❖ Аппроксимация обучающего набора, показанного на рис. 5.2, полиномиальной моделью регрессии высокой степени. Истинная функция квадратичная, но мы здесь используем только модели степени 9. Мы варьируем степень снижения весов, чтобы предотвратить переобучение модели высокой степени. (*Слева*) Выбрав λ очень большим, можно заставить модель обучить функцию вообще без наклона. Такая модель недообучена, поскольку может представить только постоянную функцию. (*В центре*) При среднем значении λ алгоритм обучения реконструирует кривую примерно правильной формы. И хотя модель способна представить функции гораздо более сложной формы, из-за снижения веса она вынуждена использовать более простые функции с малыми весовыми коэффициентами. (*Справа*) Когда λ приближается к нулю (т. е. регуляризация минимальна, и мы используем псевдообратную матрицу Мура–Пенроуза для решения недоопределенной задачи), многочлен степени 9 оказывается сильно переобученным, что мы уже видели на рис. 5.2

Вообще, мы можем регуляризовать модель, которая обучает функцию $f(\boldsymbol{x}, \boldsymbol{\theta})$, прибавив к функции стоимости штраф, называемый **регуляризатором**. В случае снижения весов регуляризатор имеет вид $\Omega(\boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{w}$. В главе 7 мы встретим много других регуляризаторов.

Выражение предпочтения одной функции перед другой – более общий способ управления емкостью модели, чем расширение или сужение пространства гипотез. Исключение функции из пространства гипотез можно трактовать как бесконечно большое «предпочтение» против этой функции.

В примере снижения весов мы явно выразили предпочтение линейным функциям с меньшими весами, включив дополнительный член в минимизируемый критерий. Есть много других способов – явных и неявных – отдать предпочтение другим решениям. Все такие подходы называются **регуляризацией**. Регуляризация – это любая модификация алгоритма обучения, предпринятая с целью уменьшить его ошибку обобщения, не уменьшая ошибку обучения. Регуляризация – одна из важнейших тем машинного обучения, с которой соперничать может только оптимизация.

Из теоремы об отсутствии бесплатных завтраков вытекает, что не существует наилучшего алгоритма машинного обучения и, в частности, наилучшей формы регуляризации. Мы должны выбирать ту форму регуляризации, которая отвечает конкретной решаемой задаче. Философия глубокого обучения вообще и этой книги в частности состоит в том, что имеется широкий круг задач (например, все интеллектуальные задачи, выполняемые людьми), которые можно эффективно решить, применяя весьма универсальные формы регуляризации.

5.3. Гиперпараметры и контрольные наборы

У большинства алгоритмов машинного обучения имеются гиперпараметры, управляющие поведением алгоритма. Значения гиперпараметров не отыскиваются самим алгоритмом (хотя можно построить вложенную процедуру, в которой один алгоритм обучения будет находить гиперпараметры для другого).

В примере полиномиальной регрессии на рис. 5.2 один гиперпараметр: степень многочлена, он играет роль гиперпараметра емкости. Другой пример гиперпараметра – значение λ, контролирующее силу снижения весов.

Иногда настройку делают гиперпараметром, а не обучают, потому что оптимизация слишком сложна. Но чаще причина в том, что бессмысленно обучать гиперпараметр на обучающем наборе. Это относится ко всем гиперпараметрам, управляющим емкостью модели. При попытке обучить их на обучающем наборе всегда выбиралась бы максимально возможная емкость модели, что приводило бы к переобучению (см. рис. 5.3). Например, мы всегда можем взять многочлен высокой степени и положить коэффициент снижения весов $\lambda = 0$ – аппроксимация обучающего набора при этом будет лучше, чем для многочлена более низкой степени и положительного λ.

Для решения этой проблемы нам нужен **контрольный набор** примеров, которых алгоритм не видел в процессе обучения.

Выше мы обсуждали, как можно использовать тестовый набор примеров, выбранных из того же распределения, что и обучающий, для оценки ошибки обобщения уже после завершения процесса обучения. Важно, что тестовые примеры вообще не используются для принятия каких-либо решений о модели, в т. ч. и касающихся ее гиперпараметров. Поэтому ни один пример из тестового набора не должен входить в контрольный. Следовательно, контрольный набор всегда формируется из *обучаю-*

щих данных. Точнее, мы разбиваем обучающие данные на два непересекающихся подмножества. Одно из них используется для обучения параметров, а другое является контрольным набором и служит для оценки ошибки обобщения во время обучения или по его завершении; это позволяет подстраивать гиперпараметры. Подмножество данных, используемое для обучения параметров, по-прежнему называется обучающим набором, несмотря на возможную путаницу с более широким набором данных, фигурирующим в процессе всего обучения. Подмножество данных, используемое для управления выбором гиперпараметров, называется контрольным набором. Обычно 80% всех обучающих данных резервируется для обучения, а 20% – для контроля. Поскольку контрольный набор используется для «обучения» гиперпараметров, ошибка на контрольном наборе является заниженной оценкой ошибки обобщения, но, как правило, на меньшую величину, чем ошибка обучения. По завершении оптимизации гиперпараметров ошибку обобщения можно оценить на тестовом наборе.

На практике, когда один и тот же тестовый набор на протяжении многих лет используется для оценки качества различных алгоритмов, а научное сообщество прилагает максимум усилий, чтобы превысить предыдущие достижения именно на этом наборе, дело заканчивается тем, что результаты, измеренные на тестовом наборе, оказываются чрезмерно оптимистичными. То, что раньше считалось эталоном, устаревает и перестает отражать истинное качество обученной системы на реальных данных. Тогда сообщество постепенно переходит на новый (обычно более амбициозный и объемный) эталонный набор данных.

5.3.1. Перекрестная проверка

Разделение набора данных на фиксированные обучающий и тестовый становится проблематичным, если в тестовом наборе оказывается слишком мало данных. Малый тестовый набор – причина статистической недостоверности в оценке средней ошибки тестирования, из-за которой трудно утверждать, что алгоритм A работает лучше алгоритма B на конкретной задаче.

Если набор данных насчитывает сотни тысяч и более примеров, то особой проблемы не возникает. Но когда набор данных мал, можно прибегнуть к альтернативным процедурам, которые позволяют использовать все примеры для оценивания средней ошибки тестирования ценой увеличения объема вычислений. Идея этих процедур – в том, чтобы повторять обучение и оценку на разных случайно выбранных подмножествах (разделениях) исходного набора данных. Наиболее распространена процедура k-групповой перекрестной проверки (алгоритм 5.1), в которой набор данных разбивается на k непересекающихся подмножеств. Ошибку тестирования можно оценить, усреднив ошибки тестирования по k испытаниям. В ходе i-го испытания i-е подмножество используется в качестве тестового набора, а остальные данные – в качестве обучающего. Проблема в том, что не существует несмещенных оценок дисперсии такой средней ошибки (Bengio and Grandvalet, 2004), но обычно используются аппроксимации.

5.4. Оценки, смещение и дисперсия

Математическая статистика дает много инструментов для достижения основной цели машинного обучения: не только решить задачу на обучающем наборе, но и обеспечить возможность обобщения. Такие фундаментальные понятия, как оценивание параметров, смещение и дисперсия, полезны для формальной характеристики обобщения, недообучения и переобучения.

5.4.1. Точечное оценивание

Точечное оценивание – это попытка найти единственное «наилучшее» предсказание интересующей величины. В общем случае интересующей величиной может быть один или несколько параметров некоторой параметрической модели, например веса в примере линейной регрессии из раздела 5.1.4, а также целая функция.

Чтобы отличить оценку параметра от истинного значения, мы будем обозначать оценку параметра $\boldsymbol{\theta}$ символом $\hat{\boldsymbol{\theta}}$.

Пусть $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ – множество m независимых и одинаково распределенных точек. **Точечной оценкой**, или **статистикой**, называется любая функция этих данных:

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}) . \tag{5.19}$$

В этом определении не требуется, чтобы g возвращала значение, близкое к истинному значению $\boldsymbol{\theta}$, ни даже чтобы область значений g совпадала со множеством допустимых значений $\boldsymbol{\theta}$. Определение точечной оценки весьма общее и предоставляет проектировщику большую свободу. Хотя в соответствии с ним оценкой можно назвать почти любую функцию, хорошей оценкой будет та, значение которой близко к истинному распределению $\boldsymbol{\theta}$, из которого выбирались обучающие данные.

Алгоритм 5.1. Алгоритм k-групповой перекрестной проверки. Применяется для оценивания ошибки обобщения алгоритма обучения A, когда имеющийся набор данных $\mathbb{D}$ слишком мал для того, чтобы простое разделение на обучающий и тестовый или обучающий и контрольный наборы могло дать точную оценку ошибки обобщения, поскольку среднее значение потери L на малом тестовом наборе может иметь высокую дисперсию. Элементами набора данных $\mathbb{D}$ являются абстрактные примеры $\boldsymbol{z}^{(i)}$ вида (вход, выход) $\boldsymbol{z}^{(i)} = (\boldsymbol{x}^{(i)}, y^{(i)})$ в случае обучения с учителем или просто вход $\boldsymbol{z}^{(i)} = \boldsymbol{x}^{(i)}$ в случае обучения без учителя. Алгоритм возвращает векторы ошибок $\boldsymbol{e}$ для каждого примера из $\mathbb{D}$, усреднение по которым считается оценкой ошибки обобщения. Ошибки на отдельных примерах можно использовать для вычисления доверительного интервала вокруг среднего (уравнение 5.47). Хотя доверительные интервалы после перекрестной проверки не очень хорошо теоретически обоснованы, на практике их все же часто используют, объявляя, что алгоритм A лучше алгоритма B, только если доверительный интервал ошибки алгоритма A лежит ниже и не пересекается с доверительным интервалом алгоритма B.

Define `KFoldXV`($\mathbb{D}$, A, L, k):
Require: $\mathbb{D}$ – набор данных, состоящий из элементов $\boldsymbol{z}^{(i)}$
Require: A – алгоритм обучения, т. е. функция, которая принимает набор данных и возвращает обученную функцию
Require: L – функция потерь, т. е. функция от обученной функции f и примера $\boldsymbol{z}^{(i)} \in \mathbb{D}$, возвращающая скаляр $\in \mathbb{R}$
Require: k – число групп
  Разбить $\mathbb{D}$ на k непересекающихся подмножеств $\mathbb{D}_i$, объединение которых равно $\mathbb{D}$
  for i от 1 до k **do**
    $f_i = A(\mathbb{D} \backslash \mathbb{D}_i)$
    for $\boldsymbol{z}^{(j)}$ из $\mathbb{D}_i$ **do**
      $e_j = L(f_i, \boldsymbol{z}^{(j)})$
    end for
  end for
  Return $\boldsymbol{e}$

Пока что примем частотный подход к статистике, т. е. будем предполагать, что истинное значение параметра $\boldsymbol{\theta}$ фиксировано, но неизвестно, а точечная оценка $\hat{\boldsymbol{\theta}}$ является функцией от этих данных. Поскольку данные порождаются случайным процессом, любая функция от них также случайная. Таким образом, $\hat{\boldsymbol{\theta}}$ – случайная величина.

Точечную оценку можно также рассматривать как оценку связи между входной и выходной величинами. Такие типы точечных оценок мы будем называть оценками функций.

Оценивание функций. Иногда нас интересует оценивание (или аппроксимация) функции. В этом случае мы пытаемся предсказать величину $\boldsymbol{y}$, зная входной вектор $\boldsymbol{x}$. Предполагается, что существует функция $f(\boldsymbol{x})$, приближенно описывающая связь между $\boldsymbol{y}$ и $\boldsymbol{x}$. Например, можно предположить, что $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\varepsilon}$, где $\boldsymbol{\varepsilon}$ обозначает часть $\boldsymbol{y}$, которую невозможно предсказать, зная $\boldsymbol{x}$. Оценивание функции сводится к аппроксимации f моделью или оценкой $\hat{f}$. Оценивание функции – по существу, то же самое, что оценивание параметра θ; $\hat{f}$ – это просто точечная оценка в пространстве функций. Примеры линейной (раздел 5.1.4) и полиномиальной (раздел 5.2) регрессий можно интерпретировать либо как оценку параметра $\boldsymbol{w}$, либо как оценку функции $\hat{f}$, отображающей $\boldsymbol{x}$ в y.

Теперь рассмотрим те свойства оценок, которые чаще всего изучаются, и обсудим, какую информацию об оценках они несут.

5.4.2. Смещение

Смещение оценки определяется следующим образом:

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}, \tag{5.20}$$

где математическое ожидание вычисляется по данным (рассматриваемым как выборка из случайной величины), а $\boldsymbol{\theta}$ – истинное значение параметра, которое определяет порождающее распределение. Оценка $\hat{\boldsymbol{\theta}}_m$ называется **несмещенной**, если $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{0}$, т. е. $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$. Оценка $\hat{\boldsymbol{\theta}}_m$ называется **асимптотически несмещенной**, если $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{0}$, т. е. $\lim_{m\to\infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Пример: распределение Бернулли. Рассмотрим множество независимых примеров $\{x^{(1)}, ..., x^{(m)}\}$, имеющих одно и то же распределение Бернулли со средним θ:

$$P(x^{(i)}, \theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}. \tag{5.21}$$

Стандартной оценкой параметра θ этого распределения является среднее значение обучающих примеров:

$$\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}. \tag{5.22}$$

Чтобы узнать, является ли эта оценка смещенной, подставим (5.22) в (5.20):

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}[\hat{\theta}_m] - \theta \tag{5.23}$$

$$= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta \tag{5.24}$$

$$= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}[x^{(i)}] - \theta \tag{5.25}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{x^{(i)}=0}^{1}(x^{(i)}\theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}) - \theta \tag{5.26}$$

$$= \frac{1}{m}\sum_{i=1}^{m}(\theta) - \theta \tag{5.27}$$

$$= \theta - \theta = 0 \tag{5.28}$$

Поскольку $\text{bias}(\hat{\theta}) = 0$, оценка $\hat{\theta}$ является несмещенной.

Пример: оценка среднего нормального распределения. Теперь рассмотрим множество независимых примеров $\{x^{(1)}, ..., x^{(m)}\}$, имеющих одно и то же нормальное распределение $p(x^{(i)}) = \mathcal{N}(x^{(i)}, \mu; \sigma^2)$, где $i \in \{1, ..., m\}$. Напомним, что функция плотности вероятности нормального распределения имеет вид:

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2}\frac{(x^{(i)}-\mu)^2}{\sigma^2}\right). \tag{5.29}$$

Стандартная оценка среднего нормального распределения называется **выборочным средним**:

$$\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m}x^{(i)}. \tag{5.30}$$

Чтобы найти смещение выборочного среднего, нужно вычислить его математическое ожидание:

$$\text{bias}(\hat{\mu}_m) = \mathbb{E}[\hat{\mu}_m] - \mu \tag{5.31}$$

$$= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}x^{(i)}\right] - \mu \tag{5.32}$$

$$= \left(\frac{1}{m}\sum_{i=1}^{m}\mathbb{E}[x^{(i)}]\right) - \mu \tag{5.33}$$

$$= \left(\frac{1}{m}\sum_{i=1}^{m}\mu\right) - \mu \tag{5.34}$$

$$= \mu - \mu = 0 \tag{5.35}$$

Таким образом, выборочное среднее – несмещенная оценка среднего значения нормального распределения.

Пример: оценки дисперсии нормального распределения. В этом примере мы сравним две оценки параметра σ^2, определяющего дисперсию нормального распределения. Нас будет интересовать, являются ли они смещенными.

Первая оценка σ^2 называется **выборочной дисперсией**:

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2, \tag{5.36}$$

где $\hat{\mu}_m$ – выборочное среднее. Нас интересует величина

$$\text{bias}[\hat{\sigma}_m^2] = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2. \tag{5.37}$$

Сначала вычислим член $\mathbb{E}[\hat{\sigma}_m^2]$:

$$\mathbb{E}[\hat{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] \tag{5.38}$$

$$= \frac{m-1}{m}\sigma^2 \tag{5.39}$$

Возвращаясь к формуле (5.37), заключаем, что смещение $\hat{\sigma}_m^2$ равно $-\sigma^2/m$. Следовательно, эта оценка смещенная.

Несмещенная оценка дисперсии выглядит так:

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2. \tag{5.40}$$

Покажем, что $\mathbb{E}[\tilde{\sigma}_m^2] = \sigma^2$.

$$\mathbb{E}[\tilde{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] \tag{5.41}$$

$$= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}_m^2] \tag{5.42}$$

$$= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) \tag{5.43}$$

$$= \sigma^2 \tag{5.44}$$

Итак, у нас есть две оценки: смещенная и несмещенная. Хотя несмещенные оценки, очевидно, желательны, не всегда они являются «наилучшими». Мы часто будем использовать смещенные оценки, которые обладают другими важными свойствами.

5.4.3. Дисперсия и стандартная ошибка

Еще одно интересное свойство оценки – насколько сильно она изменяется как функция примера. Для определения смещения мы вычисляли математическое ожидание оценки, но точно так же можем вычислить и ее дисперсию. Дисперсией оценки называется выражение:

$$\mathrm{Var}(\hat{\theta}), \tag{5.45}$$

где случайной величиной является обучающий набор. **Стандартной ошибкой** $\mathrm{SE}(\hat{\theta})$ называется квадратный корень из дисперсии.

Дисперсия, как и стандартная ошибка, оценки измеряет, как будет изменяться оценка, вычисленная по данным, при независимой повторной выборке из набора данных, генерируемого порождающим процессом. Желательные оценки – обладающие не только малым смещением, но и относительно малой дисперсией.

При вычислении любой статистики по выборке конечного размера оценка истинного значения параметра всегда недостоверна в том смысле, что, взяв другую выборку из того же распределения, мы получили бы другую статистику. Ожидаемая степень вариативности оценки – источник ошибки, который мы хотим выразить количественно.

Стандартная ошибка среднего равна

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}, \tag{5.46}$$

где σ^2 – истинная дисперсия выборки x^i. Стандартную ошибку часто оценивают с помощью оценки σ. К сожалению, ни квадратный корень из выборочной дисперсии, ни квадратный корень из несмещенной оценки дисперсии не является несмещенной оценкой стандартного отклонения. В обоих случаях оценка истинного стандартного отклонения оказывается заниженной, тем не менее та и другая используются на практике. Квадратный корень из несмещенной оценки дисперсии – менее заниженная оценка. Для больших m аппроксимация вполне приемлема.

Стандартная ошибка среднего очень полезна в экспериментах по машинному обучению. Мы часто оцениваем ошибку обобщения, вычисляя выборочное среднее ошибки на тестовом наборе. Количество примеров в тестовом наборе определяет точность оценки. Воспользовавшись центральной предельной теоремой, согласно которой среднее имеет приблизительно нормальное распределение, мы можем применить стандартную ошибку для вычисления вероятности того, что истинное математическое ожидание находится в выбранном интервале. Например, 95-процентный доверительный интервал вокруг среднего $\hat{\mu}_m$ определяется формулой

$$(\hat{\mu}_m - 1.96\mathrm{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\mathrm{SE}(\hat{\mu}_m)) \tag{5.47}$$

при нормальном распределении со средним $\hat{\mu}_m$ и дисперсией $\mathrm{SE}(\hat{\mu}_m)^2$. В экспериментах по машинному обучению принято говорить, что алгоритм A лучше алгоритма B, если верхняя граница 95-процентного доверительного интервала для ошибки алгоритма A меньше нижней границы 95-процентного доверительного интервала для ошибки алгоритма B.

Пример: распределение Бернулли. Снова рассмотрим множество независимых примеров $\{x^{(1)}, ..., x^{(m)}\}$, имеющих одно и то же распределение Бернулли (напомним, что $P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1 - \theta)^{(1-x^{(i)})}$). На этот раз нас интересует дисперсия оценки $\hat{\theta}_m = (1/m)\Sigma_{i=1}^{m} x^{(i)}$.

$$\mathrm{Var}(\hat{\theta}_m) = \mathrm{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \tag{5.48}$$

$$= \frac{1}{m^2}\sum_{i=1}^{m} \mathrm{Var}(x^{(i)}) \tag{5.49}$$

$$= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \tag{5.50}$$

$$= \frac{1}{m^2} m\theta(1-\theta) \tag{5.51}$$

$$= \frac{1}{m}\theta(1-\theta) \tag{5.52}$$

Дисперсия оценки является убывающей функцией от m, количества примеров в наборе данных. Это общее свойство популярных оценок, к которому мы еще вернемся при обсуждении состоятельности в разделе 5.4.5.

5.4.4. Поиск компромисса между смещением и дисперсией для минимизации среднеквадратической ошибки

Смещение и дисперсия – два источника ошибки оценки. Смещение измеряет ожидаемое отклонение от истинного значения функции или параметра, а дисперсия – это мера отклонения от ожидаемого значения оценки в произвольной выборке данных.

Что, если имеются две оценки, у одной из которых больше смещение, а у другой дисперсия? Какую выбрать? Представим, к примеру, что мы хотим аппроксимировать функцию на рис. 5.2, и на выбор предлагаются только две модели: с большой дисперсией и с большим смещением. Какую предпочесть?

Самый распространенный подход к выбору компромиссного решения – воспользоваться перекрестной проверкой. Эмпирически продемонстрировано, что перекрестная проверка дает отличные результаты во многих реальных задачах. Можно также сравнить **среднеквадратическую ошибку** (MSE) обеих оценок:

$$\text{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] \tag{5.53}$$

$$= \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m). \tag{5.54}$$

MSE измеряет ожидаемое расхождение (в смысле квадрата ошибки) между оценкой и истинным значением параметра θ. Из формулы (5.54) видно, что в вычислении MSE участвует как смещение, так и дисперсия. Желательной является оценка с малой MSE, именно такие оценки держат под контролем и смещение, и дисперсию.

Соотношение между смещением и дисперсией тесно связано с возникающими в машинном обучении понятиями емкости модели, недообучения и переобучения. Если ошибка обобщения измеряется посредством MSE (и тогда смещение и дисперсия становятся важными компонентами ошибки обобщения), то увеличение емкости влечет за собой повышение дисперсии и снижение смещения. Это показано на рис. 5.6, где мы снова видим U-образную кривую зависимости ошибки обобщения от емкости.

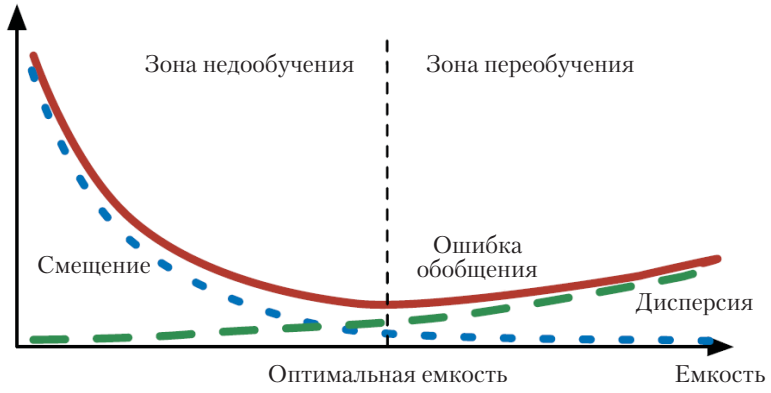

Рис. 5.6 ❖ С ростом емкости (ось x) смещение (пунктирная линия) уменьшается, а дисперсия (штриховая линия) увеличивается, в результате чего получается еще одна U-образная кривая для ошибки обобщения (жирная линия). В какой-то точке на оси x емкость оптимальна, слева от этой точки имеет место недообучение, справа – переобучение. Это соотношение аналогично соотношению между емкостью, недообучением и переобучением, которое рассматривалось в разделе 5.2 и на рис. 5.3

5.4.5. Состоятельность

До сих пор мы обсуждали свойства различных оценок для обучающего набора фиксированного размера. Обычно нас интересует также поведение оценки по мере роста этого размера. В частности, мы хотим, чтобы при увеличении числа примеров точечные оценки сходились к истинным значениям соответствующих параметров. Формально это записывается в виде:

$$\text{plim}_{m\to\infty}\hat{\theta}_m = \theta. \tag{5.55}$$

Символом plim обозначается сходимость в смысле вероятности, т. е. для любого $\varepsilon > 0$ $P(|\hat{\theta}_m - \theta| > \varepsilon) \to 0$ при $m \to \infty$. Условие (5.55) называется **состоятельностью**. Иногда его называют слабой состоятельностью, понимая под сильной состоятельностью **сходимость почти наверное** $\hat{\theta}$ к θ. Говорят, что последовательность случайных величин $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ... сходится к $\boldsymbol{x}$ почти наверное, если $\text{p}(\lim_{m\to\infty}\mathbf{x}^{(m)} = \boldsymbol{x}) = 1$.

Состоятельность гарантирует, что смещение оценки уменьшается с ростом числа примеров. Однако обратное неверно – из асимптотической несмещенности не вытекает состоятельность. Рассмотрим, к примеру, оценивание среднего μ нормального распределения $\mathcal{N}(x; \mu, \sigma^2)$ по набору данных, содержащему m примеров: $\{x^{(1)}, ..., x^{(m)}\}$. Можно было бы взять в качестве оценки первый пример: $\hat{\theta} = x^{(1)}$. В таком случае $\mathbb{E}(\hat{\theta}_m) = \theta$, поэтому оценка является несмещенной вне зависимости от того, сколько примеров мы видели. Отсюда, конечно, следует, что оценка асимптотически несмещенная. Но она не является состоятельной, т. к. *неверно*, что $\hat{\theta}_m \to \theta$ при $m \to \infty$.

5.5. Оценка максимального правдоподобия

Мы познакомились с определениями нескольких оценок и проанализировали их свойства. Но откуда взялись эти оценки? Хотелось бы не угадывать функцию, которая могла бы оказаться хорошей оценкой, и затем анализировать смещение и дисперсию, а располагать каким-то общим принципом, позволяющим выводить функции, являющиеся хорошими оценками для разных моделей.

Чаще всего для этой цели применяется принцип максимального правдоподобия.

Рассмотрим множество m примеров $\mathbb{X} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$, независимо выбираемых из неизвестного порождающего распределения $p_{\text{data}}(\mathbf{x})$.

Обозначим $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ параметрическое семейство распределений вероятности над одним и тем же пространством, индексированное параметром $\boldsymbol{\theta}$. Иными словами, $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ отображает произвольную конфигурацию $\boldsymbol{x}$ на вещественное число, дающее оценку истинной вероятности $p_{\text{data}}(\boldsymbol{x})$.

Тогда оценка максимального правдоподобия для $\boldsymbol{\theta}$ определяется формулой:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}), \tag{5.56}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.57}$$

Такое произведение большого числа вероятностей по ряду причин может быть неудобно. Например, оно подвержено потере значимости. Для получения эквивалентной, но более удобной задачи оптимизации заметим, что взятие логарифма правдоподобия не изменяет arg max, но преобразует произведение в сумму:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.58}$$

Поскольку arg max не изменяется при умножении функции стоимости на константу, мы можем разделить правую часть на m и получить выражение в виде математического ожидания относительно эмпирического распределения $\hat{p}_{\text{data}}$, определяемого обучающими данными:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\text{x}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x};\boldsymbol{\theta}). \tag{5.59}$$

Один из способов интерпретации оценки максимального правдоподобия состоит в том, чтобы рассматривать ее как минимизацию расхождения Кульбака–Лейблера между эмпирическим распределением $\hat{p}_{\text{data}}$, определяемым обучающим набором, и модельным распределением. Расхождение Кульбака–Лейблера определяется формулой

$$D_{\text{KL}}(\hat{p}_{\text{data}} \| p_{\text{model}}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}} [\log \hat{p}_{\text{data}}(\boldsymbol{x}) - \log p_{\text{model}}(\boldsymbol{x})]. \tag{5.60}$$

Первый член разности в квадратных скобках зависит только от порождающего данные процесса, но не от модели. Следовательно, при обучении модели, минимизирующей расхождение КЛ, мы должны минимизировать только величину

$$-\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}} [\log p_{\text{model}}(\boldsymbol{x})], \tag{5.61}$$

а это, конечно, то же самое, что максимизация величины (5.59).

Минимизация расхождения КЛ в точности соответствует минимизации перекрестной энтропии между распределениями. Многие авторы употребляют термин «перекрестная энтропия» для обозначения исключительно отрицательного логарифмического правдоподобия распределения Бернулли или softmax, но это неправильно. Любая функция потерь, содержащая отрицательное логарифмическое правдоподобие, является перекрестной энтропией между эмпирическим распределением, определяемым обучающим набором, и распределением, определяемым моделью. Например, среднеквадратическая ошибка – перекрестная энтропия между эмпирическим распределением и гауссовой моделью.

Таким образом, мы видим, что максимальное правдоподобие – это попытка совместить модельное распределение с эмпирическим распределением $\hat{p}_{\text{data}}$. В идеале мы хотели бы совпадения с истинным порождающим распределением p_{data}, но непосредственного доступа к нему у нас нет.

Хотя оптимальное значение $\boldsymbol{\theta}$ не зависит от того, максимизируем мы правдоподобие или минимизируем расхождение КЛ, значения целевых функций различны. При разработке программ мы часто называем то и другое минимизацией функции стоимости. В таком случае поиск максимального правдоподобия становится задачей минимизации отрицательного логарифмического правдоподобия (ОЛП), или, что эквивалентно, минимизации перекрестной энтропии. Взгляд на максимальное правдоподобие как на минимальное расхождение КЛ в этом случае становится полезен, потому что известно, что минимум расхождения КЛ равен нулю. А отрицательное логарифмическое правдоподобие может принимать отрицательные значения при вещественных $\boldsymbol{x}$.

5.5.1. Условное логарифмическое правдоподобие и среднеквадратическая ошибка

Оценку максимального правдоподобия легко обобщить для оценивания условной вероятности $P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$, чтобы предсказывать $\mathbf{y}$ при известном $\mathbf{x}$. На самом деле это вполне типичная ситуация, лежащая в основе большинства алгоритмов обучения

с учителем. Если $\boldsymbol{X}$ представляет все входы, а $\boldsymbol{Y}$ – все наблюдаемые выходы, то оценка условного максимального правдоподобия равна

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}). \tag{5.62}$$

Если предположить, что все примеры независимы и одинаково распределены, то это выражение можно представить в виде суммы:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.62}$$

Пример: линейная регрессия и максимальное правдоподобие. Линейную регрессию (раздел 5.1.4) можно интерпретировать как нахождение оценки максимального правдоподобия. Ранее мы описывали линейную регрессию как алгоритм, который обучается порождать значение $\hat{y}$ по входным данным $\boldsymbol{x}$. Отображение $\boldsymbol{x}$ в $\hat{y}$ выбирается, так чтобы минимизировать среднеквадратическую ошибку – критерий, взятый более-менее произвольно. Теперь мы рассмотрим линейную регрессию с точки зрения оценки максимального правдоподобия. Будем считать, что цель не в том, чтобы вернуть одно предсказание $\hat{y}$, а чтобы построить модель, порождающую условное распределение $p(y \mid \boldsymbol{x})$. Можно представить себе, что в бесконечно большом обучающем наборе мы могли бы встретить несколько обучающих примеров с одним и тем же значением $\boldsymbol{x}$, но с разными значениями y. Цель алгоритма обучения теперь – аппроксимировать $p(y \mid \boldsymbol{x})$, подогнав его под все эти разные значения y, совместимые с $\boldsymbol{x}$. Для вывода такого же алгоритма линейной регрессии, как и раньше, определим $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; \hat{y}(\boldsymbol{x}; \boldsymbol{w}), \sigma^2)$. Функция $\hat{y}(\boldsymbol{x}; \boldsymbol{w})$ дает предсказание среднего значения нормального распределения. В этом примере мы предполагаем, что дисперсия фиксирована и равна константе σ^2, задаваемой пользователем. Мы увидим, что при таком выборе функциональной формы $p(y \mid \boldsymbol{x})$ нахождение оценки максимального правдоподобия сводится к тому же алгоритму обучения, что был разработан ранее. Поскольку предполагается, что примеры независимы и одинаково распределены, то условное логарифмическое правдоподобие (формула 5.63) записывается в виде:

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.64}$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{\left\| \hat{y}^{(i)} - y^{(i)} \right\|^2}{2\sigma^2}, \tag{5.65}$$

где $\hat{y}^{(i)}$ – результат линейной регрессии для i-го примера $\boldsymbol{x}^{(i)}$, а m – число обучающих примеров. Сравнивая логарифмическое правдоподобие со среднеквадратической ошибкой

$$\mathrm{MSE}_{\mathrm{train}} = \frac{1}{m} \sum_{i=1}^{m} \left\| \hat{y}^{(i)} - y^{(i)} \right\|^2, \tag{5.66}$$

мы сразу же видим, что максимизация логарифмического правдоподобия относительно $\boldsymbol{w}$ дает ту же оценку параметров $\boldsymbol{w}$, что минимизация среднеквадратической ошибки. Значения этих критериев различны, но положение оптимума совпадает. Это служит обоснованием использования среднеквадратической ошибки в качестве оценки максимального правдоподобия. Как мы увидим, оценка максимального правдоподобия обладает рядом желательных свойств.

5.5.2. Свойства максимального правдоподобия

Главное достоинство оценки максимального правдоподобия заключается в том, что с точки зрения скорости сходимости она является асимптотически наилучшей оценкой, когда количество примеров $m \rightarrow \infty$.

При определенных условиях оценка максимального правдоподобия обладает свойством состоятельности (см. раздел 5.4.5), т. е. когда число обучающих примеров стремится к бесконечности, оценка максимального правдоподобия параметра сходится к истинному значению этого параметра. Вот эти условия:

- истинное распределение p_{data} принадлежит семейству модельных распределений $p_{\text{model}}(\cdot; \boldsymbol{\theta})$. В противном случае никакая оценка не сможет реконструировать p_{data};
- истинное распределение p_{data} соответствует ровно одному значению $\boldsymbol{\theta}$. В противном случае оценка максимального правдоподобия сможет реконструировать p_{data}, но не сможет определить, какое значение $\boldsymbol{\theta}$ было использовано в порождающем процессе.

Существуют и другие индуктивные принципы, помимо оценки максимального правдоподобия, и многие из них обладают свойством состоятельности. Однако состоятельные оценки могут различаться по **статистической эффективности**, т. е. некая оценка может давать меньшую ошибку обобщения при фиксированном числе примеров m или, эквивалентно, требовать меньше примеров для получения заданного уровня ошибки обобщения.

Статистическая эффективность обычно изучается в **параметрическом случае** (например, в линейной регрессии), когда наша цель – оценить значение параметра (в предположении, что выявить истинный параметр возможно), а не значение функции. Для измерения степени близости к истинному значению параметра используется ожидаемая среднеквадратическая ошибка, описывающая квадрат разности между оценкой и истинным значением параметра, причем математическое ожидание вычисляется по m обучающим примерам, взятым из порождающего распределения. Эта параметрическая среднеквадратическая ошибка убывает с ростом m, и для больших m справедливо неравенство Крамера–Рао (Rao, 1945; Cramér, 1946), показывающее, что ни для какой состоятельной оценки среднеквадратическая ошибка не может быть меньше, чем для оценки максимального правдоподобия.

По этим причинам (состоятельность и эффективность) оценка максимального правдоподобия часто считается предпочтительной в машинном обучении. Если количество примеров настолько мало, что есть угроза переобучения, то можно применить стратегии регуляризации, например снижение весов, которые дают смещенный вариант оценки максимального правдоподобия с меньшей дисперсией.

5.6. Байесовская статистика

До сих пор мы обсуждали **частотные статистики** и подходы, основанные на оценивании единственного значения $\boldsymbol{\theta}$, которое затем использовалось для всех предсказаний. Другой подход состоит в том, чтобы делать предсказание, рассматривая все возможные значения $\boldsymbol{\theta}$. Это область **байесовской статистики**.

Как было сказано в разделе 5.4.1, частотный подход предполагает, что истинное значение $\boldsymbol{\theta}$ фиксировано, хотя и неизвестно, а точечная оценка $\hat{\boldsymbol{\theta}}$ – случайная величина ввиду того, что она является функцией набора данных (который считается случайным).

Байесовский подход к статистике совершенно иной. Вероятность в этом случае отражает степень уверенности в наших знаниях. Набор данных доступен прямому наблюдению и, следовательно, не является случайным. С другой стороны, истинный параметр $\boldsymbol{\theta}$ неизвестен или недостоверен и потому представляется случайной переменной.

Еще до наблюдения данных мы представляем свои знания про $\boldsymbol{\theta}$ в виде априорного распределения вероятности $p(\boldsymbol{\theta})$. В машинном обучении априорное распределение обычно берется достаточно широким (имеющим большую энтропию), что отражает высокую степень неопределенности значения $\boldsymbol{\theta}$ до наблюдения. Например, априори можно предположить, что $\boldsymbol{\theta}$ равномерно распределен в некоторой конечной области. Во многих априорных распределениях предпочтение, наоборот, отдается «более простым» решениям (например, коэффициентам с меньшими абсолютными величинами или функции, близкой к постоянной).

Рассмотрим теперь набор примеров $\{x^{(1)}, ..., x^{(m)}\}$. Можно реконструировать влияние данных на наши гипотезы про $\boldsymbol{\theta}$, объединив правдоподобие данных $p(x^{(1)}, ..., x^{(m)} \mid \boldsymbol{\theta})$ с априорным распределением посредством правила Байеса:

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})}. \tag{5.67}$$

В тех ситуациях, где обычно применяется байесовское оценивание, в качестве априорного представления берется относительно равномерное или нормальное распределение с большой энтропией, а наблюдение данных, как правило, приводит к апостериорному распределению с меньшей энтропией, сосредоточенному в окрестности вероятных значений параметров.

По сравнению с оценкой максимального правдоподобия, у байесовской оценки есть два важных отличия. Во-первых, в подходе на основе максимального правдоподобия предсказания делаются с использованием точечной оценки $\boldsymbol{\theta}$, а в байесовском – с использованием полного распределения $\boldsymbol{\theta}$. Например, после наблюдения m примеров предсказанное распределение следующего примера $x^{(m+1)}$ описывается формулой

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) d\boldsymbol{\theta}. \tag{5.68}$$

Здесь каждое значение $\boldsymbol{\theta}$ с положительной плотностью вероятности вносит вклад в предсказание следующего примера с весом, равным самой апостериорной плотности. Если после наблюдения примеров $\{x^{(1)}, ..., x^{(m)}\}$ мы все еще не уверены в значении $\boldsymbol{\theta}$, то эта неопределенность включается прямо в предсказания.

В разделе 5.4 мы обсуждали, как в частотном подходе решается вопрос о неопределенности данной точечной оценки $\boldsymbol{\theta}$ путем вычисления ее дисперсии. Дисперсия оценки – это суждение о том, как могла бы измениться оценка в случае другой выборки наблюдаемых данных. Байесовский подход дает другой ответ на вопрос о неопределенности оценки – просто проинтегрировать по ней, и это хорошо защищает от переобучения. Этот интеграл – не что иное, как применение законов вероятности, поэтому байесовский подход легко обосновать, тогда как частотный механизм построения оценки основан на довольно-таки произвольном решении свести все знания, содержащиеся в наборе данных, к одной точечной оценке.

Второе важное различие между байесовским подходом к оцениванию и подходом на основе максимального правдоподобия обусловлено вкладом байесовского апри-

орного распределения. Влияние этого распределения состоит в сдвиге плотности вероятности в сторону тех областей пространства параметров, которые априори предпочтительны. На практике априорное распределение часто выражает предпочтение более простым или более гладким моделям. Критики байесовского подхода называют априорное распределение источника субъективных суждений влияющим на предсказания.

Обычно байесовские методы обобщаются гораздо лучше при ограниченном объеме обучающих данных, но становятся вычислительно более накладными, когда количество обучающих примеров велико.

Пример: байесовская линейная регрессия. Рассмотрим байесовский подход к обучению параметров линейной регрессии. Мы хотим обучить линейное отображение входного вектора $\boldsymbol{x} \in \mathbb{R}^n$ для предсказания скалярного значения $y \in \mathbb{R}$. Предсказание параметризуется вектором $\boldsymbol{w} \in \mathbb{R}^n$:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}. \tag{5.69}$$

Если дано множество m обучающих примеров $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, то предсказание y по всему обучающему набору можно выразить в виде

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})}\boldsymbol{w}. \tag{5.70}$$

Предполагая нормальное условное распределение $\boldsymbol{y}^{(\text{train})}$, имеем

$$p(\boldsymbol{y}^{(\text{train})} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{w}) = \mathcal{N}(\boldsymbol{y}^{(\text{train})}; \boldsymbol{X}^{(\text{train})}\boldsymbol{w}, \boldsymbol{I}), \tag{5.71}$$

$$\propto \exp(-{}^1\!/_2(\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})}\boldsymbol{w})^\top(\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})}\boldsymbol{w})), \tag{5.72}$$

где используется стандартное для записи среднеквадратической ошибки предположение о том, что дисперсия y равна 1. В дальнейшем мы во избежание громоздкости будем вместо $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$ писать просто $(\boldsymbol{X}, \boldsymbol{y})$.

Для определения апостериорного распределения вектора параметров модели $\boldsymbol{w}$ нужно сначала задать априорное распределение. Оно должно отражать наши наивные представления о ценности параметров. Иногда выразить априорные представления в терминах параметров модели трудно или неестественно, но на практике мы обычно предполагаем довольно широкое распределение, выражающее высокую степень неопределенности $\boldsymbol{\theta}$. Для вещественных параметров часто в качестве априорного берут нормальное распределение.

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \propto \exp(-{}^1\!/_2(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)), \tag{5.73}$$

где $\boldsymbol{\mu}_0$ и $\boldsymbol{\Lambda}_0$ – средний вектор априорного распределения и ковариационная матрица соответственно[1].

При таком задании априорного распределения мы теперь можем перейти к определению **апостериорного** распределения параметров модели:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w}), \tag{5.74}$$

$$\propto \exp(-{}^1\!/_2(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})) \exp(-{}^1\!/_2(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)) \tag{5.75}$$

$$\propto \exp(-{}^1\!/_2(-2\boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top\boldsymbol{\Lambda}_0^{-1}\boldsymbol{w} - 2\boldsymbol{\mu}_0^\top\boldsymbol{\Lambda}_0^{-1}\boldsymbol{w})). \tag{5.76}$$

---

[1] Если нет причин использовать конкретную структуру ковариационной матрицы, то обычно берется диагональная $\boldsymbol{\Lambda}_0 = \text{diag}(\boldsymbol{\lambda}_0)$.

Определим $\boldsymbol{\Lambda}_m = (\boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\Lambda}_0^{-1})^{-1}$ и $\boldsymbol{\mu}_m = \boldsymbol{\Lambda}_m(\boldsymbol{X}^\top \boldsymbol{y} + \boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0)$. При так определенных новых переменных апостериорное распределение можно записать в виде нормального:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp(-{}^1/_2(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) + {}^1/_2\boldsymbol{\mu}_m^\top \boldsymbol{\Lambda}_m^{-1}\boldsymbol{\mu}_m), \tag{5.77}$$

$$\propto \exp(-{}^1/_2(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m)). \tag{5.78}$$

Все члены, не включающие вектор параметров $\boldsymbol{w}$, опущены; они подразумеваются в силу того факта, что распределение должно быть нормировано, так чтобы интеграл оказался равен 1. Как нормируется многомерное нормальное распределение, показано в формуле (3.23).

Изучение этого апостериорного распределения позволяет составить интуитивное представление о поведении байесовского вывода. В большинстве случаев мы задаем $\boldsymbol{\mu}_0$ равным $\mathbf{0}$. Если положить $\boldsymbol{\Lambda}_0 = (1/\alpha)\boldsymbol{I}$, то μ_m дает ту же оценку $\boldsymbol{w}$, что и частотная линейная регрессия со снижением веса $\alpha\boldsymbol{w}^\top\boldsymbol{w}$. Одно отличие заключается в том, что байесовская оценка не определена, если α равно 0 – запрещается начинать процесс байесовского обучения с бесконечно широким априорным распределением $\boldsymbol{w}$. Но есть и более важное отличие – байесовская оценка дает ковариационную матрицу, показывающую, насколько вероятны все значения $\boldsymbol{w}$, а не только оценку μ_m.

5.6.1. Оценка апостериорного максимума (MAP)

Хотя принципиальный подход состоит в том, чтобы делать предсказания, используя полное байесовское апостериорное распределение параметра $\boldsymbol{\theta}$, часто все же желательно иметь одну точечную оценку. Одна их причин – тот факт, что операции, включающие байесовское апостериорное распределение для большинства интересных моделей, как правило, вычислительно неразрешимы, а точечная оценка предлагает разрешимую аппроксимацию. Вместо того чтобы просто возвращать оценку максимального правдоподобия, мы можем все-таки воспользоваться некоторыми преимуществами байесовского подхода, разрешив априорному распределению влиять на выбор точечной оценки. Один из рациональных способов сделать это – взять оценку апостериорного максимума (MAP). Это точка, в которой достигается максимальная апостериорная вероятность (или максимальная плотность вероятности в более распространенном случае непрерывной величины $\boldsymbol{\theta}$).

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}). \tag{5.79}$$

В правой части мы видим знакомый член $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$ – стандартное логарифмическое правдоподобие, а также слагаемое $\log p(\boldsymbol{\theta})$, соответствующее априорному распределению.

В качестве примера рассмотрим модель линейной регрессии с нормальным априорным распределением весов $\boldsymbol{w}$. Если это распределение $\mathcal{N}(\boldsymbol{w}; \mathbf{0}, (1/\lambda)\boldsymbol{I}^2)$, то член $\log p(\theta)$ в формуле (5.79) пропорционален знакомому штрафу в виде снижения весов $\lambda\boldsymbol{w}^\top\boldsymbol{w}$ плюс член, не зависящий от $\boldsymbol{w}$ и не влияющий на процесс обучения. Таким образом, байесовский вывод MAP с нормальным априорным распределением весов соответствует снижению весов.

Байесовский вывод MAP, как и полный байесовский вывод, обладает тем преимуществом, что задействует информацию, содержащуюся в априорном распределении, но отсутствующую в обучающих данных. Эта дополнительная информация помогает

уменьшить дисперсию точечной оценки MAP (по сравнению с оценкой максимального правдоподобия). Однако за это приходится расплачиваться увеличенным смещением.

Многие регуляризированные стратегии оценивания, в частности оценку максимального правдоподобия со снижением весом, можно интерпретировать как MAP-аппроксимацию байесовского вывода. Эта точка зрения применима, когда процедура регуляризации сводится к прибавлению дополнительного члена к целевой функции, который соответствует $\log p(\boldsymbol{\theta})$. Не все регуляризующие штрафы совместимы с байесовским выводом. Например, возможны члены-регуляризаторы, не являющиеся логарифмом распределения вероятности. Бывает и так, что член-регуляризатор зависит от данных, что, конечно, недопустимо для априорного распределения.

Байесовский вывод MAP предлагает простой способ проектирования сложных, но все же допускающих интерпретацию регуляризующих членов. Например, более хитроумный штрафной член можно получить, взяв в качестве априорного смесь нормальных распределений, а не единственное такое распределение (Nowlan and Hinton, 1992).

5.7. Алгоритмы обучения с учителем

В разделе 5.1.3 было сказано, что алгоритм обучения с учителем – это такой алгоритм, в котором обучающий набор примеров содержит не только входные данные $\boldsymbol{x}$, но и ассоциированные с ними выходные данные $\boldsymbol{y}$. Во многих случаях выходные данные трудно собрать автоматически, поэтому их должен предоставить человек – «учитель», однако сам термин применяется и тогда, когда метки входных данных собирались автоматически.

5.7.1. Вероятностное обучение с учителем

Большинство алгоритмов обучения с учителем, рассматриваемых в этой книге, основано на оценивании распределения вероятности $p(y \mid \boldsymbol{x})$. Это можно сделать, просто воспользовавшись оценкой максимального правдоподобия, чтобы найти наилучший вектор параметров $\boldsymbol{\theta}$ для параметрического семейства распределений $p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

Мы уже видели, что линейная регрессия соответствует семейству

$$p(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y; \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{I}). \tag{5.80}$$

Мы можем обобщить линейную регрессию на случай классификации, определив другое семейство распределений вероятности. Если имеются два класса: 0 и 1, то нужно лишь задать вероятность каждого класса. При этом вероятность класса 1 однозначно определяет вероятность класса 0, поскольку сумма обоих значений должна быть равна 1.

Нормальное распределение вещественных чисел, с которым мы имели дело в задаче линейной регрессии, параметризуется величиной среднего. Допустимо любое значение среднего. Распределение бинарной величины несколько сложнее, потому что ее среднее должно заключаться между 0 и 1. Решить эту проблему можно: например, воспользоваться логистической сигмоидой, которая «втискивает» значение линейной функции в интервал (0, 1), и интерпретировать это значение как вероятность:

$$p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}). \tag{5.81}$$

Этот подход называется **логистической регрессией** (довольно странное название, поскольку модель используется не для регрессии, а для классификации).

В задаче линейной регрессии мы смогли найти оптимальные веса, решив нормальные уравнения. С логистической регрессией дело обстоит сложнее. В этом случае не существует замкнутой формулы для нахождения оптимальных весов. Их нужно искать путем максимизации функции логарифмического правдоподобия. Сделать это можно, найдя минимум отрицательного логарифмического правдоподобия методом градиентного спуска.

Ту же стратегию можно применить практически к любой задаче обучения с учителем, выписав параметрическое семейство условных распределений вероятности подходящих входных и выходных величин.

5.7.2. Метод опорных векторов

Из методов, оказавших наибольшее влияние на обучение с учителем, следует упомянуть прежде всего метод опорных векторов (support vector machine – SVM) (Boser et al., 1992; Cortes and Vapnik, 1995). Эта модель похожа на логистическую регрессию тем, что описывается линейной функцией $\boldsymbol{w}^\top \boldsymbol{x} + b$. Но, в отличие от логистической регрессии, метод опорных векторов вычисляет не вероятности, а только класс. SVM предсказывает положительный класс, если значение $\boldsymbol{w}^\top \boldsymbol{x} + b$ положительно, и отрицательный – если оно отрицательно.

Одно из основных новшеств, принесенных методом опорных векторов, – это **kernel trick** (трюк с ядром). Идея этого приема заключается в том, что многие алгоритмы машинного обучения можно выразить исключительно в терминах скалярного произведения примеров. Например, можно показать, что линейную функцию, используемую в методе опорных векторов, можно переписать в виде:

$$\boldsymbol{w}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)}, \tag{5.82}$$

где $\boldsymbol{x}^{(i)}$ – обучающий пример, а α – вектор коэффициентов. Записав алгоритм обучения в таком виде, мы сможем заменить $\boldsymbol{x}$ результатом заданной функции признаков $\phi(\boldsymbol{x})$, а скалярное произведение – функцией $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$, которая называется **ядром**. Оператор $\cdot$ представляет скалярное произведение, аналогичное $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$. Для некоторых пространств признаков использовать скалярное произведение векторов в чистом виде невозможно. В некоторых бесконечномерных пространствах нужны другие виды скалярного произведения, например основанные на интегрировании, а не на суммировании. Разработка полной теории таких вариантов скалярного произведения выходит за рамки книги. Заменив скалярные произведения вычислением ядра, мы можем делать предсказания, пользуясь функцией

$$f(\boldsymbol{x}) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}). \tag{5.83}$$

Эта функция нелинейно зависит от $\boldsymbol{x}$, но соотношение между $\phi(\boldsymbol{x})$ и $f(\boldsymbol{x})$ линейное. Также линейным является соотношение между α и $f(\boldsymbol{x})$. Основанная на ядре функция в точности эквивалентна предварительной обработке путем применения $\phi(\boldsymbol{x})$ ко всем входным данным с последующим обучением линейной модели в новом преобразованном пространстве.

Трюк с ядром полезен по двум причинам. Во-первых, он позволяет обучать модели, нелинейно зависящие от $\boldsymbol{x}$, применяя методы выпуклой оптимизации, о которых

точно известно, что они сходятся эффективно. Это возможно, потому что мы считаем $\boldsymbol{\phi}$ фиксированным и оптимизируем только α, т. е. алгоритм оптимизации может считать, что решающая функция линейна, но в другом пространстве. Во-вторых, ядерная функция k часто допускает реализацию, значительно более эффективную с вычислительной точки зрения, чем наивное построение двух векторов $\boldsymbol{\phi}(\boldsymbol{x})$ и явное вычисление их скалярного произведения.

В некоторых случаях вектор $\boldsymbol{\phi}(\boldsymbol{x})$ может быть даже бесконечномерным, что при наивном подходе вылилось бы в бесконечно длинное вычисление. Часто $k(\boldsymbol{x}, \boldsymbol{x}')$ – нелинейная, но разрешимая функция $\boldsymbol{x}$, даже если $\boldsymbol{\phi}(\boldsymbol{x})$ неразрешима. В качестве примера бесконечномерного пространства признаков с разрешимым ядром построим отображение $\boldsymbol{\phi}(x)$, когда признаки $\boldsymbol{x}$ – неотрицательные целые числа. Это отображение возвращает вектор, содержащий $\boldsymbol{x}$ единиц, за которыми следует бесконечное число нулей. Тогда ядерная функция $k(x, x^{(i)}) = \min(x, x^{(i)})$ в точности эквивалентна скалярному произведению таких бесконечномерных векторов.

Самым распространенным является гауссово ядро

$$k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2\boldsymbol{I}), \tag{5.84}$$

где $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ – стандартная функция плотности нормального распределения. Это ядро называется также **радиально-базисной функцией** (RBF), потому что его значение убывает вдоль прямых в пространстве $\boldsymbol{v}$, исходящих из $\boldsymbol{u}$. Гауссово ядро соответствует скалярному произведению в бесконечномерном пространстве, но описание этого пространства не так тривиально, как в примере ядра min над целыми числами.

Можно считать, что ядро Гаусса реализует некое **сравнение с образцом**. Обучающий пример $\boldsymbol{x}$, с которым ассоциирована метка y, становится образцом класса y. Если тестовая точка $\boldsymbol{x}'$ близка к $\boldsymbol{x}$ в смысле евклидова расстояния, то гауссово ядро дает большой отклик, показывающий, что $\boldsymbol{x}'$ очень похож на образец $\boldsymbol{x}$. Тогда модель назначает большой вес ассоциированной обучающей метке y. Итоговое предсказание объединяет много обучающих меток, взвешенных в соответствии с похожестью на соответствующие обучающие примеры.

Метод опорных векторов – не единственный алгоритм, который можно улучшить с помощью трюка с ядром. Категория алгоритмов, в которых используется трюк с ядром, называется **ядерными методами** (Williams and Rasmussen, 1996; Schölkopf et al., 1999).

Главный недостаток ядерных методов – тот факт, что сложность вычисления решающей функции линейно зависит от числа обучающих примеров, поскольку i-й пример вносит член $\alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$ в решающую функцию. В методе опорных векторов эта проблема сглаживается тем, что обучаемый вектор $\boldsymbol{\alpha}$ содержит в основном нули. Тогда для классификации нового примера требуется вычислить ядерную функцию только для обучающих примеров с ненулевыми α_i. Эти обучающие примеры и называются **опорными векторами**.

Для ядерных методов характерна также высокая стоимость обучения при большом наборе данных. Мы вернемся к этому вопросу в разделе 5.9. Ядерные методы с ядрами общего вида плохо обобщаются. Почему это так, мы объясним в разделе 5.11. Глубокое обучение в своем современном воплощении предназначено для преодоления этих ограничений ядерных методов. Ренессанс глубокого обучения начался с работы Hinton et al. (2006), где было продемонстрировано, что нейронная сеть способна превзойти метод SVM с радиально-базисным ядром на эталонном наборе данных MNIST.

5.7.3. Другие простые алгоритмы обучения с учителем

Мы уже встречались с другим невероятностным алгоритмом обучения с учителем – регрессией методом ближайшего соседа. В общем случае для классификации или регрессии используется семейство методов с k ближайшими соседями. Будучи непараметрическим алгоритмом обучения, метод k ближайших соседей не ограничен фиксированным числом параметров. Обычно считается, что этот алгоритм вообще не имеет параметров, а реализует простую функцию от обучающих данных. На самом деле в нем даже нет настоящего этапа обучения. Просто на этапе тестирования, желая породить выход y для тестового примера $\boldsymbol{x}$, мы ищем в обучающем наборе $\boldsymbol{X}$ k ближайших соседей $\boldsymbol{x}$, а затем возвращаем среднее соответствующих им меток y. Эта идея работает для любого вида обучения с учителем, при условии что можно определить понятие средней метки. В случае классификации усреднять можно по унитарным кодовым векторам $\boldsymbol{c}$, в которых $c_y = 1$ и $c_i = 0$ для всех остальных i. Среднее по таким векторам можно интерпретировать как распределение вероятности классов. Алгоритм k ближайших соседей, являясь непараметрическим, может достигать очень высокой емкости. Предположим, к примеру, что имеются задача многоклассовой классификации и мера производительности с бинарной функцией потерь. В таком случае метод одного ближайшего соседа сходится к удвоенной байесовской частоте ошибок, когда число обучающих примеров стремится к бесконечности. Превышение над байесовской ошибкой связано с тем, что мы случайным образом выбираем одного из равноудаленных соседей. Если число обучающих примеров бесконечно, то у всех тестовых точек $\boldsymbol{x}$ будет бесконечно много соседей из обучающего набора на нулевом расстоянии. Если бы алгоритм разрешал соседям проголосовать, а не выбирал случайного соседа, то процедура сходилась бы к байесовской частоте ошибок. Высокая емкость алгоритма k ближайших соседей позволяет получить высокую верность, если имеется большой обучающий набор. Но за это приходится расплачиваться высокой стоимостью вычислений, а при малом обучающем наборе алгоритм плохо обобщается. Одно из слабых мест алгоритма k ближайших соседей – неумение понять, что один признак является более отличительным, чем другой. Возьмем, к примеру, задачу регрессии, в которой $\boldsymbol{x} \in \mathbb{R}^{100}$ выбирается из изотропного нормального распределения, но результат зависит только от переменной x_1. Предположим еще, что результат попросту совпадает с этим признаком, т. е. $y = x_1$ во всех случаях. Регрессия методом ближайшего соседа не сможет уловить эту простую закономерность. Ближайший сосед большинства точек $\boldsymbol{x}$ будет определяться по большому числу признаков от x_2 до x_{100}, а не только по признаку x_1. Поэтому на небольшом обучающем наборе результат будет, по существу, случайным.

Еще один тип алгоритма обучения, также разбивающий пространство входов на области, каждая из которых описывается отдельными параметрами, – **решающее дерево** (Breiman et al., 1984) и его многочисленные варианты. На рис. 5.7 показано, что с каждым узлом решающего дерева ассоциирована область пространства входов, и внутренние узлы разбивают эту область на две части – по одной для каждого дочернего узла (обычно рассекая параллельно оси). Таким образом, пространство входов делится на непересекающиеся области, взаимно однозначно соответствующие листовым узлам. Обычно каждый листовый узел сопоставляет каждой входной точке в своей области один и тот же выход. Для обучения решающих деревьев применяются специальные алгоритмы, выходящие за рамки этой книги. Алгоритм обучения можно

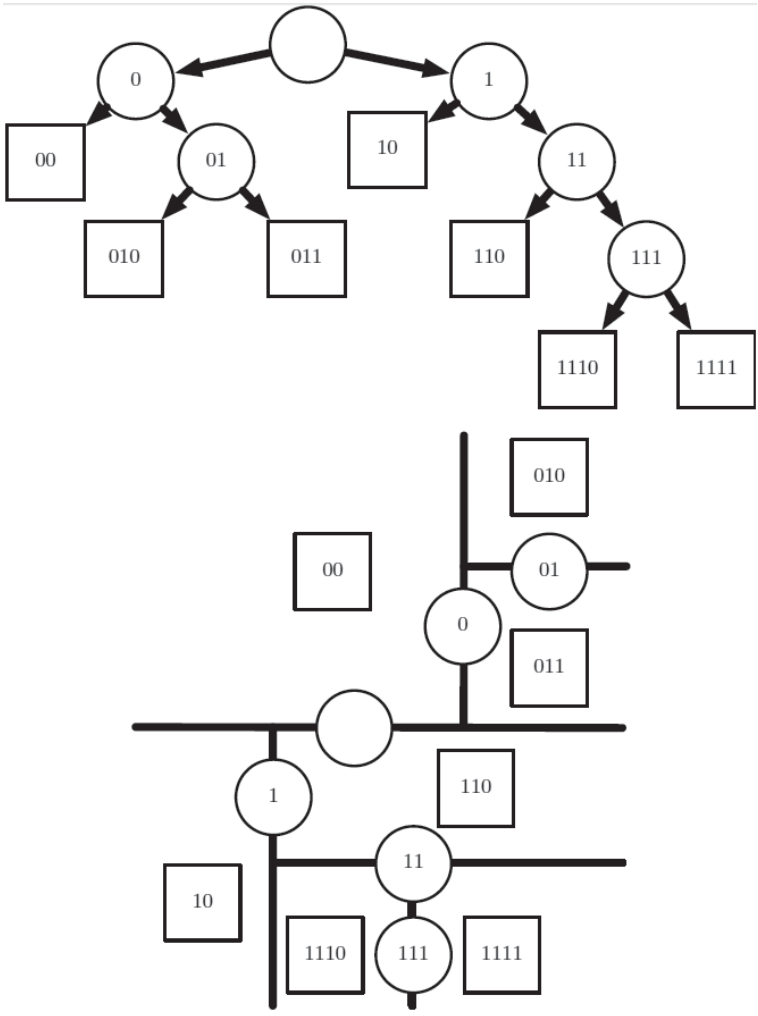

Рис. 5.7 ❖ Описание работы решающего дерева. (*Вверху*) Каждый узел дерева выбирает, кому отправить входной пример: левому потомку (0) или правому (1). Внутренние узлы изображены кружочками, листовые – квадратиками. Каждый узел обозначается строкой нулей и единиц, описывающей его позицию в дереве. Строка получается дописыванием одного разряда к идентификатору родительского узла (0 – слева от родителя, 1 – справа от родителя). (*Внизу*) Дерево разбивает пространство на области. На рисунке приведен пример разбиения плоскости $\mathbb{R}^2$. На этой плоскости нарисованы узлы дерева, через каждый внутренний узел проходит разделительная линия, показывающая, как он классифицирует примеры, а листовые узлы находятся в центре множества попадающих в них примеров. В результате получается кусочно-постоянная функция, сопоставляющая свой «кусок» каждому листовому узлу. В каждом листовом узле должен находиться хотя бы один пример, так что решающее дерево не может обучить функцию, имеющую больше локальных максимумов, чем число обучающих примеров.

считать непараметрическим, если ему разрешено обучать дерево произвольного размера, хотя обычно решающие деревья регуляризируются с помощью ограничений на размер, так что на практике модель становится параметрической. В том виде, в каком решающие деревья обычно используются – с осепараллельным разделением в узлах и постоянным выходом в каждом узле, – они с трудом решают некоторые задачи, с которыми легко справляется даже логистическая регрессия. Например, если имеется двухклассовая задача и класс положителен тогда и только тогда, когда $x_2 > x_1$, то решающая граница не параллельна оси. Поэтому решающему дереву придется аппроксимировать решающую границу большим количеством узлов – реализовать ступенчатую функцию с осепараллельными шагами, которая постоянно колеблется по обе стороны истинной решающей функции.

Как мы видели, у предикторов на основе ближайших соседей и решающих деревьев много ограничений. Тем не менее эти алгоритмы обучения полезны, когда мы стеснены в вычислительных ресурсах. Мы также можем интуитивно оценить поведение более сложных алгоритмов обучения, размышляя о сходстве и различии между ними и эталонами: ближайшими соседями и решающими деревьями. Дополнительный материал об алгоритмах обучения с учителем см. в работах Murphy (2012), Bishop (2006), Hastie et al. (2001) и других учебниках по машинному обучению.

5.8. Алгоритмы обучения без учителя

Напомним (раздел 5.1.3), что алгоритмом обучения без учителя называется алгоритм, который получает на входе только «признаки» без какой-либо подсказки со стороны учителя. Различие между обучением с учителем и без учителя определено не формально и жестко, потому что не существует объективного критерия, что считать признаком, а что – меткой. Неформально под обучением без учителя понимают попытки извлечь информацию из распределения, выборка из которого не была вручную аннотирована человеком. Этот термин обычно ассоциируется с оцениванием плотности, выборкой примеров из распределения, очисткой выбранных из некоторого распределения данных от шума, нахождением многообразия, рядом с которым располагаются данные, или кластеризацией данных – выделением групп похожих примеров.

Классическая задача обучения без учителя – найти «наилучшее» представление данных. Под «наилучшим» можно понимать разные вещи, но в общем случае ищется представление, которое сохраняет как можно больше информации о $\boldsymbol{x}$, соблюдая при этом ограничения, призванные сделать представление проще или понятнее, чем само $\boldsymbol{x}$.

Определить более простое представление можно многими способами. Три самых распространенных: представления меньшей размерности, разреженные представления и независимые представления. В случае понижения размерности мы пытаемся спрессовать как можно больше информации об $\boldsymbol{x}$ в представление меньшего размера. Разреженное представление (Barlow, 1989; Olshausen and Field, 1996; Hinton and Ghahramani, 1997) – это погружение набора данных в представление, где большинству входов соответствуют нули. В этом случае размерность представления обычно увеличивается, так чтобы наличие многих нулей не приводило к отбрасыванию слишком большого объема информации. В результате получается представление, в котором данные распределены в основном вдоль осей. Независимые представления – это попытка разделить источники вариативности в истинном распределении данных, чтобы измерения представления оказались статистически независимыми.

Разумеется, эти критерия не исключат друг друга. Представления низкой размерности часто дают элементы, между которыми меньше зависимостей или зависимости более слабые, чем в исходных данных высокой размерности. Так происходит, потому что один из способов уменьшить размер представления – найти и устранить избыточность. Чем больше избыточности удается устранить, тем больше степень сжатия и тем меньше потери информации.

Понятие представления – одна из центральных тем глубокого обучения, а вместе с тем и книги. В этом разделе мы приведем несколько простых алгоритмов обучения представлений. В совокупности эти примеры показывают, как вышеупомянутые критерии выглядят на практике. В последующих главах будут описаны дополнительные алгоритмы обучения представлений, где эти критерии развиваются в разных направлениях или вводятся новые.

5.8.1. Метод главных компонент

В разделе 2.12 мы видели, что метод главных компонент позволяет сжимать данные. PCA можно рассматривать как алгоритм обучения без учителя, который ищет представление данных, основанное на двух из трех описанных выше критериев простого представления. PCA обучает представление, имеющее меньшую размерность, чем исходное. Кроме того, в этом представлении между элементами нет линейной корреляции. Это первый шаг к нахождению представления со статистически независимыми элементами. Чтобы полностью избавиться от зависимостей, алгоритм обучения должен удалить также нелинейные связи между переменными.

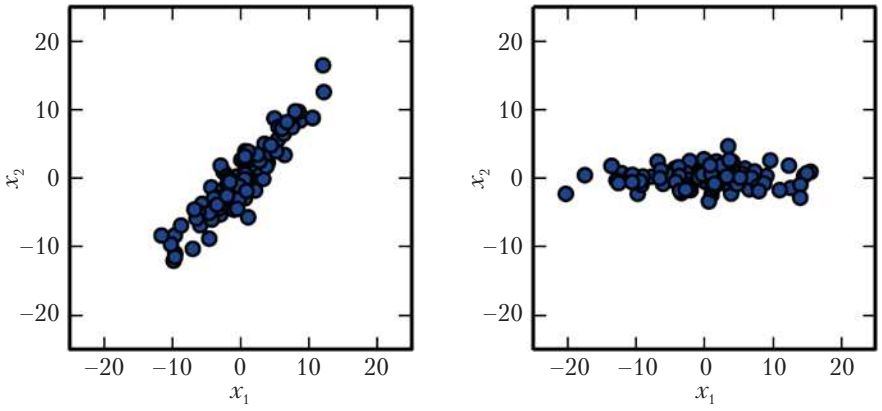

Рис. 5.8 ❖ Результатом PCA являются проекции на прямые, параллельные направлению наибольшей дисперсии. Они становятся осями нового пространства. (*Слева*) Исходные данные содержат выборку из $\boldsymbol{x}$. В этом пространстве дисперсия может быть максимальна вдоль прямых, не параллельных осям. (*Справа*) После преобразования $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$ наибольшее изменение сосредоточено вдоль оси z_1. Направление второй по величине дисперсии стало осью z_2

PCA находит ортогональное линейное преобразование, переводящее входные данные $\boldsymbol{x}$ в представление $\boldsymbol{z}$, как показано на рис. 5.8. В разделе 2.12 мы видели, что можно обучить одномерное представление, наилучшим образом реконструирующее

исходные данные (в смысле среднеквадратической ошибки), и что это представление на самом деле соответствует первой главной компоненте данных. Следовательно, мы можем использовать PCA как простой и эффективный метод понижения размерности, который сохраняет столько присутствующей в данных информации, сколько возможно (опять же на основе измерения ошибки реконструкции по методу наименьших квадратов). Ниже мы изучим, как представление PCA устраняет корреляцию из исходного представления $\boldsymbol{X}$.

Рассмотрим матрицу плана $\boldsymbol{X}$ размера $m \times n$. Будем предполагать, что математическое ожидание данных $\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$. Если это не так, центрирования легко добиться, вычтя среднее из всех примеров на этапе предварительной обработки.

Несмещенная выборочная ковариационная матрица, ассоциированная с $\boldsymbol{X}$, определяется по формуле

$$\mathrm{Var}[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X}. \tag{5.85}$$

PCA находит представление (посредством линейного преобразования) $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x}$, для которого $\mathrm{Var}[\boldsymbol{z}]$ – диагональная матрица.

В разделе 2.12 мы видели, что главные компоненты матрицы плана $\boldsymbol{X}$ определяются собственными векторами $\boldsymbol{X}^\top \boldsymbol{X}$. Таким образом,

$$\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{W}\boldsymbol{\Lambda}\boldsymbol{W}^\top. \tag{5.86}$$

В этом разделе мы по-другому подойдем к выводу главных компонент. Их можно получить также путем сингулярного разложения (SVD). Точнее, это правые сингулярные векторы X. Чтобы убедиться в этом, предположим, что $\boldsymbol{W}$ – правые сингулярные векторы в разложении $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top$. Тогда исходное уравнение собственных векторов можно переписать в базисе $\boldsymbol{W}$:

$$\boldsymbol{X}^\top \boldsymbol{X} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top = \boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top. \tag{5.87}$$

Разложение SVD полезно для доказательства того, что PCA приводит к диагональной матрице $\mathrm{Var}[z]$. Применяя сингулярное разложение $\boldsymbol{X}$, мы можем выразить дисперсию $\boldsymbol{X}$ в виде:

$$\mathrm{Var}[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X} \tag{5.88}$$

$$= \frac{1}{m-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top \tag{5.89}$$

$$= \frac{1}{m-1}\boldsymbol{W}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top \tag{5.90}$$

$$= \frac{1}{m-1}\boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top, \tag{5.91}$$

где используется тот факт, что $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$, поскольку матрица $\boldsymbol{U}$ в сингулярном разложении по определению ортогональная. Отсюда следует, что ковариационная матрица $\boldsymbol{z}$ диагональная, что и требовалось доказать:

$$\mathrm{Var}[\boldsymbol{z}] = \frac{1}{m-1}\boldsymbol{Z}^\top \boldsymbol{Z} \tag{5.92}$$

$$= \frac{1}{m-1}\boldsymbol{W}^{\top}\boldsymbol{X}^{\top}\boldsymbol{X}\boldsymbol{W} \tag{5.93}$$

$$= \frac{1}{m-1}\boldsymbol{W}^{\top}\boldsymbol{W}\boldsymbol{\Sigma}^{2}\boldsymbol{W}^{\top}\boldsymbol{W} \tag{5.94}$$

$$= \frac{1}{m-1}\boldsymbol{\Sigma}^{2}. \tag{5.95}$$

На этот раз мы воспользовались тем, что $\boldsymbol{W}^{\top}\boldsymbol{W} = \boldsymbol{I}$ – опять же по определению сингулярного разложения.

Проведенный анализ показывает, что представление, полученное в результате проецирования данных $\boldsymbol{x}$ на $\boldsymbol{z}$ посредством линейного преобразования $\boldsymbol{W}$, имеет диагональную ковариационную матрицу ($\boldsymbol{\Sigma}^2$). А отсюда сразу вытекает, что взаимная корреляция отдельных элементов $\boldsymbol{z}$ равна нулю.

Это свойство PCA – преобразовывать данные в представление с взаимно не коррелированными элементами – очень важно. Оно дает простой пример представления, пытающегося *разделить неизвестные факторы вариативности* данных. В случае PCA разделение сводится к поиску такого вращения пространства входных данных (описываемого матрицей $\boldsymbol{W}$), которое делает главные оси дисперсии базисом нового пространства представления $\boldsymbol{z}$.

Хотя корреляция – важная категория зависимостей между элементами данных, при обучении представлений нам также интересно разделить более сложные виды зависимостей между признаками, не исчерпывающимися простыми линейными преобразованиями.

5.8.2. Кластеризация методом *k* средних

Еще один пример простого алгоритма обучения представлений дает кластеризация методом k средних. Этот алгоритм разбивает обучающий набор на k кластеров, содержащих близкие примеры. Можно считать, что этот алгоритм вырабатывает k-мерный унитарный кодовый вектор $\boldsymbol{h}$, представляющий вход $\boldsymbol{x}$. Если $\boldsymbol{x}$ принадлежит i-му кластеру, то $h_i = 1$, а все остальные элементы $\boldsymbol{h}$ равны 0.

Унитарный код, порождаемый кластеризацией методом k средних, – пример разреженного представления, поскольку для каждого входного примера большинство элементов представления равно нулю. Впоследствии мы разработаем другие алгоритмы, вырабатывающие более гибкие разреженные представления, в которых ненулевыми может быть несколько элементов. Унитарный код – крайний случай разреженного представления, в котором утрачены многие преимущества распределенных представлений. Но все же у него есть некоторые статистические достоинства (он естественно передает идею о том, что все примеры из одного кластера похожи друг на друга), а также чисто вычислительное преимущество: все представление можно выразить одним целым числом.

В начале работы алгоритма инициализируется k различных центроидов $\{\boldsymbol{\mu}^{(1)}, ..., \boldsymbol{\mu}^{(k)}\}$, а затем поочередно выполняются два шага до достижения сходимости. На одном шаге каждый обучающий пример относится к i-му кластеру, где i – индекс ближайшего центроида $\boldsymbol{\mu}^{(i)}$. На другом шаге каждому центроиду $\boldsymbol{\mu}^{(i)}$ присваивается значение, равное среднему всех отнесенных к i-му кластеру примеров $\boldsymbol{x}^{(j)}$.

Одной из проблем является тот факт, что задача кластеризации некорректно поставлена в том смысле, что не существует однозначного критерия, позволяющего су-

дить, в какой мере найденная кластеризация данных соответствует реальности. Мы можем измерить такие свойства кластеризации, как среднее евклидово расстояние от центроида кластера до его членов. Это позволит сказать, насколько хорошо можно реконструировать обучающие данные по отнесению к кластерам. Но мы не знаем, насколько хорошо отнесение примеров к кластерам отражает свойства реального мира. Более того, может существовать несколько разных способов кластеризации, одинаково хорошо описывающих реальный мир. Быть может, мы надеялись найти кластеризацию, относящуюся к одному признаку, а нашли совсем другую, не менее хорошую кластеризацию, не имеющую ни малейшего отношения к нашей задаче. Предположим, к примеру, что мы выполнили два разных алгоритма кластеризации для набора данных, содержащего изображения красных грузовиков, красных легковушек, серых грузовиков и серых легковушек. Если мы попросим каждый алгоритм найти два кластера, то один может найти кластер грузовиков и кластер легковушек, а другой – кластеры красных и серых транспортных средств. Допустим, есть еще и третий алгоритм, умеющий сам определять число кластеров. Он может отнести примеры к четырем кластерам: красные легковушки, красные грузовики, серые легковушки, серые грузовики. Новая кластеризация улавливает оба атрибута, но теряет информацию о сходстве. Красные легковушки не попали ни в кластер серых легковушек, ни в кластер серых грузовиков. Результат этого алгоритма ничего не говорит о том, что красные легковушки больше похожи на серые легковушки, чем на серые грузовики. Они отличаются и от того, и от другого, но это все, что мы знаем.

Эти проблемы иллюстрируют причины, по которым распределенное представление может оказаться лучше унитарного. В распределенном представлении можно было бы хранить два атрибута для каждого транспортного средства: цвет и тип (грузовик или легковушка). По-прежнему не ясно, какое распределенное представление оптимально (как алгоритм обучения мог бы узнать, что нас интересует цвет и тип, а не, скажем, производитель и возраст машины?), но хранение нескольких атрибутов, по крайней мере, позволяет алгоритму не гадать, какой единственный атрибут нас интересует, и дает возможность измерять сходство между объектами более детально, сравнивая многие атрибуты, а не просто проверяя совпадение одного единственного.

5.9. Стохастический градиентный спуск

Почти всё глубокое обучение основано на одном очень важном алгоритме: **стохастическом градиентном спуске** (СГС). Это обобщение алгоритма градиентного спуска, описанного в разделе 4.3.

В машинном обучении постоянно возникает одна и та же проблема: большой обучающий набор необходим для более качественного обобщения, но обучение на нем обходится дорого с точки зрения вычислений.

Функция стоимости, применяемая в алгоритмах машинного обучения, часто представляется в виде суммы по всем примерам некоторой функции потерь, определенной для одного примера. Так, отрицательное условное логарифмическое правдоподобие обучающих данных можно записать в виде:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \tag{5.96}$$

где L – потеря на одном примере $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

Для таких аддитивных функций потерь метод градиентного спуска вычисляет выражение

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \tag{5.97}$$

Вычислительная сложность этой операции составляет $O(m)$. Если обучающий набор содержит миллиарды примеров, то время одного шага вычисления градиента становится недопустимо большим.

Идея метода СГС состоит в том, что градиент – это математическое ожидание, и, следовательно, его можно оценить по небольшому множеству примеров. Точнее, на каждом шаге алгоритма мы можем взять **мини-пакет** (minibatch) – небольшую равномерную выборку из обучающего набора $\mathbb{B} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}\}$. Размер мини-пакета m' обычно составляет от одной до нескольких сотен. Важно, что m', как правило, фиксируется, т. е. не изменяется с ростом размера обучающего набора m. Мы можем аппроксимировать многомиллиардный обучающий набор, производя вычисления только на сотне примеров.

Оценку градиента дает следующее выражение:

$$\boldsymbol{g} = \frac{1}{m'}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \tag{5.98}$$

в котором используются только примеры из мини-пакета $\mathbb{B}$. Затем алгоритм стохастического градиентного спуска следует в направлении градиента:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \boldsymbol{g}, \tag{5.99}$$

где ε – скорость обучения.

Вообще говоря, градиентный спуск часто считался медленным или ненадежным методом. В прошлом применение градиентного спуска к задачам невыпуклой оптимизации рассматривали как авантюризм или беспринципность. Сегодня мы знаем, что модели машинного обучения, описанные в части II, прекрасно работают после обучения методом градиентного спуска. Пусть и не гарантируется, что алгоритм оптимизации найдет хотя бы локальный минимум за разумное время, зачастую он все же находит очень малое значение функции стоимости достаточно быстро, для того чтобы считаться полезным.

Стохастический градиентный спуск имеет важные применения и за пределами глубокого машинного обучения. Это основной способ обучения больших линейных моделей на очень больших наборах данных. Для модели фиксированного размера стоимость одного шага СГС не зависит от размера обучающего набора m. На практике мы часто увеличиваем размер модели по мере роста размера обучающего набора, но это необязательно. Количество шагов до достижения сходимости обычно возрастает с ростом размера обучающего набора. Но когда m стремится к бесконечности, модель в конечном итоге сходится к наилучшей возможной ошибке тестирования еще до того, как СГС проверил каждый пример из обучающего набора. Дальнейшее увеличение m не увеличивает время обучения, необходимое для достижения наилучшей ошибки тестирования. С этой точки зрения можно сказать, что асимптотическая стоимость обучения модели методом СГС как функция m имеет порядок $O(1)$.

До изобретения глубокого обучения основным способом обучения нелинейных моделей была комбинация трюка с ядром и линейной модели. Для многих ядерных

алгоритмов обучения нужно было построить матрицу $m \times m$ $G_{i,j} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$. Сложность построения такой матрицы равна $O(m^2)$, что, очевидно, неприемлемо для наборов данных с миллиардами примеров. Начиная с 2006 года интерес к глубокому обучению в академических кругах поначалу был обусловлен тем, что, по сравнению с конкурирующими алгоритмами, оно лучше обобщается на новые примеры при наборе данных среднего размера – с десятками тысяч примеров. Но вскоре после этого в отрасли пробудился дополнительный интерес, поскольку глубокое обучение давало масштабируемый способ обучения нелинейных моделей на больших наборах данных.

Метод стохастического градиентного спуска и его многочисленные усовершенствования подробно описаны в главе 8.

5.10. Построение алгоритма машинного обучения

Почти все алгоритмы глубокого обучения можно описать как варианты одного простого рецепта: комбинация набора данных, функции стоимости, процедуры оптимизации и модели.

Например, алгоритм линейной регрессии – это комбинация набора данных, состоящего из $\boldsymbol{X}$ и $\boldsymbol{y}$, функции стоимости

$$J(\boldsymbol{w}, b) = -\mathbb{E}_{\mathbf{x}, \mathrm{y} \sim \hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(y \mid \boldsymbol{x}), \tag{5.100}$$

спецификации модели $p_{\mathrm{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(y; \boldsymbol{x}^{\top}\boldsymbol{w} + b, 1)$ и в большинстве случаев алгоритма оптимизации, который путем решения нормальных уравнений находит точки, где градиент функции стоимости обращается в 0.

Осознав, что любой из этих компонентов можно заменить – как правило, независимо от остальных, – мы можем получить широкий спектр алгоритмов.

Типичная функция стоимости включает, по меньшей мере, один член, благодаря которому в процессе обучения производится статистическое оценивание. Чаще всего в роли функции стоимости выступает отрицательное логарифмическое правдоподобие, поэтому ее минимизация дает оценку максимального правдоподобия.

Функция стоимости может включать и дополнительные члены, например для регуляризации. Так, к функции стоимости линейной регрессии можно прибавить поощрение за снижение весов:

$$J(\boldsymbol{w}, b) = \lambda \|\boldsymbol{w}\|_2^2 - \mathbb{E}_{\mathbf{x}, \mathrm{y} \sim \hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(y \mid \boldsymbol{x}). \tag{5.101}$$

И в этом случае возможна оптимизация в замкнутой форме.

Если перейти от линейных моделей к нелинейным, то для большинства функций стоимости оптимизация в замкнутой форме уже невозможна. Придется выбрать какую-то процедуру численной оптимизации, например градиентный спуск.

Рецепт построения алгоритма обучения путем комбинирования модели, функции стоимости и алгоритма оптимизации подходит для обучения как с учителем, так и без оного. Линейная регрессия дает пример обучения с учителем. А для обучения без учителя нужно будет определить набор данных, содержащий только $\boldsymbol{X}$, и предоставить походящую функцию стоимости и модель. Например, для получения первого вектора методом PCA можно задать такую функцию потерь:

$$J(\boldsymbol{w}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\mathrm{data}}} \|\boldsymbol{x} - r(\boldsymbol{x}; \boldsymbol{w})\|_2^2 \tag{5.102}$$

и модель, в которой $\boldsymbol{w}$ имеет единичную норму, а в качестве функции реконструкции используется $r(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}\boldsymbol{w}$.

В некоторых случаях функцию стоимости невозможно вычислить из-за высокой стоимости вычислений. Но и тогда мы можем минимизировать ее приближенно, применяя итеративную численную оптимизацию, при условии что существует какой-то способ аппроксимации ее градиентов.

В большинстве алгоритмов машинного обучения используется именно этот рецепт, хотя не всегда это очевидно с первого взгляда. Если алгоритм кажется ни что не похожим или спроектирован вручную, обычно его можно понять, рассматривая как оптимизацию в особом случае. Для некоторых моделей, например решающих деревьев и k средних, требуются специальные оптимизации, потому что их функции стоимости имеют плоские участки, из-за которых минимизация градиентными методами не годится. Осознание того, что большинство алгоритмов машинного обучения можно описать таким рецептом, позволяет рассматривать различные алгоритмы как результат систематизации методов для решения сходных задач, которые работают по сходным причинам, а не как длинный список алгоритмов, у каждого из которых свое обоснование.

5.11. Проблемы, требующие глубокого обучения

Простые алгоритмы машинного обучения, описанные в этой главе, хорошо работают для самых разных задач. Однако они потерпели неудачу при решении центральных проблем ИИ, в частности распознавании речи и объектов.

Среди стимулов для разработки глубокого обучения была в том числе и неспособность традиционных алгоритмов добиться обобщаемости на таких задачах ИИ.

В этом разделе мы покажем, что трудность проблемы обобщения на новые примеры экспоненциально возрастает при работе с многомерными данными и что механизмов обобщения, работающих в традиционном машинном обучении, недостаточно для обучения сложных функций в многомерных пространствах, поскольку стоимость вычислений становится слишком высока. Глубокое обучение призвано преодолеть эти и другие препятствия.

5.11.1. Проклятие размерности

Многие алгоритмы машинного обучения резко усложняются, когда размерность данных велика. Это явление называется **проклятием размерности**. Особенно неприятно, что количество возможных конфигураций множества переменных возрастает экспоненциально с увеличением числа переменных.

Проклятие размерности возникает в разных разделах информатики, но особенно часто в машинном обучении.

Один из аспектов проклятия размерности связан со статистикой. На рис. 5.9 видно, что статистическая проблема возникает из-за того, что число возможных конфигураций $\boldsymbol{x}$ гораздо больше, чем число обучающих примеров. Чтобы разобраться, в чем тут дело, рассмотрим пространство входов, организованное в виде сетки, как на рисунке. Пространство низкой размерности можно описать небольшим числом ячеек сетки, по большей части занятых данными. При обобщении на новый пример мы обычно можем сказать, что делать, просто исследовав обучающие примеры, находящиеся в той же ячейке, что и новый. Например, в качестве оценки плотности вероятности в некоторой точке $\boldsymbol{x}$ мы можем просто вернуть число обучающих примеров в том же

единичном объеме, что и $\boldsymbol{x}$, поделенное на общее число обучающих примеров. Чтобы классифицировать пример, мы возвращаем класс большинства обучающих примеров в той же ячейке. В случае регрессии мы можем усреднить значения, наблюдавшиеся для примеров в этой ячейке. Но как быть с ячейками, для которых мы не видели ни одного примера? Поскольку в пространстве высокой размерности количество конфигураций огромно – гораздо больше числа примеров, – в большинстве ячеек сетки обучающих примеров нет. И как можно сказать что-то осмысленное о новых конфигурациях? В большинстве традиционных алгоритмов машинного обучения просто предполагается, что выход в новой точке должен быть примерно таким же, как в ближайшей обучающей точке.

Рис. 5.9 ❖ С увеличением размерности данных (слева направо) количество представляющих интерес конфигураций растет экспоненциально. (*Слева*) В одномерном случае имеется одна переменная, и для нее всего 10 интересных областей. При достаточном числе примеров, попадающих в каждую область (на рисунке область соответствует одной клетке), от алгоритма обучения легко добиться правильного обобщения. Самый прямолинейный способ – оценить значение метки в каждой области с возможной интерполяцией между соседними областями). (*В центре*) В двумерном случае уже труднее различить 10 разных значений каждой переменной. Нам приходится учитывать 10×10 = 100 областей, и, чтобы их все покрыть, нужно, по крайней мере, столько же примеров. (*Справа*) В трехмерном случае число областей возрастает до 103 = 1000, и соответственно растет число примеров. Если имеется d измерений и нужно различать v значений вдоль каждой оси, то потребуется $O(v^d)$ областей и примеров. Это и есть проклятие размерности. Рисунок любезно предоставил Николас Чападос

5.11.2. Регуляризация для достижения локального постоянства и гладкости

Чтобы алгоритм хорошо обобщался, необходимо иметь априорное представление о том, какого рода функцию он должен обучить. Мы видели, что эти априорные предположения выражаются явно в виде распределения вероятности параметров модели. Неформально мы можем также полагать, что априорные знания непосредственно влияют на саму функцию и лишь опосредованно на параметры, считая это результатом связи между параметрами и функцией. Кроме того, неформально априорные предположения неявно выражаются в виде выбора алгоритма, смещенного в пользу определенного класса функций, хотя такое смещение, возможно, и не выражено (а иногда его попросту невозможно выразить) в терминах распределения вероятности, описывающего степень уверенности в выборе той или иной функции.

Из самых распространенных неявных «априорных предположений» отметим **априорное предположение о гладкости**, или **априорное предположение о локальном постоянстве**. То и другое означает, что обучаемая функция не должна сильно изменяться в небольшой области.

Обобщаемость многих простых алгоритмов опирается исключительно на это априорное предположение, и потому они плохо масштабируются в условиях статистических проблем, присущих задачам ИИ. Ниже в этой книге мы увидим, что глубокое обучение вводит дополнительные (явные и неявные) априорные предположения, чтобы уменьшить ошибку обобщения на сложных задачах. А сейчас объясним, почему одного предположения о гладкости недостаточно.

Существует много способов явно или неявно выразить априорное предположение о том, что обучаемая функция должна быть гладкой или локально постоянной. Все они направлены на то, чтобы заставить процесс обучения находить функцию f^*, удовлетворяющую условию

$$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \varepsilon) \tag{5.103}$$

для большинства конфигураций $\boldsymbol{x}$ и небольшого изменения ε. Иными словами, если мы знаем хороший ответ для входа $\boldsymbol{x}$ (например, если $\boldsymbol{x}$ – помеченный обучающий пример), то этот ответ, вероятно, будет хорошим и в окрестности $\boldsymbol{x}$. Если в некоторой окрестности есть несколько хороших ответов, то можно скомбинировать их (путем какой-то формы усреднения или интерполяции) и получить ответ, согласующийся с большинством вариантов.

Крайнее проявление подхода на основе локального постоянства – алгоритмы обучения на основе k ближайших соседей. Такие предикторы действительно постоянны в области, содержащей все точки $\boldsymbol{x}$ с одним и тем же множеством k ближайших соседей из обучающего набора. Для $k = 1$ число различимых областей не может превышать количество обучающих примеров.

Если алгоритм k ближайших соседей формирует выход, копируя близкие обучающие примеры, то большинство ядерных методов выполняет интерполяцию меток близких примеров. Важным классом ядер является семейство **локальных ядер**, для которых $k(\boldsymbol{u}, \boldsymbol{v})$ велико, если $\boldsymbol{u} = \boldsymbol{v}$, и убывает мере роста расстояния между $\boldsymbol{u}$ и $\boldsymbol{v}$. Локальное ядро можно представлять себе как функцию сходства, которая производит сравнение с образцом, измеряя сходство между тестовым примером $\boldsymbol{x}$ и каждым обучающим примером $\boldsymbol{x}^{(i)}$. На развитие глубокого обучения в немалой степени повлияло изучение ограничений локального сравнения с образцом и того, как глубокие модели могут добиться успеха там, где этот метод терпит неудачу (Bengio et al., 2006b).

Решающие деревья также подвержены ограничениям обучения, основанного только на предположении о гладкости, поскольку разделяют пространство на столько областей, сколько имеется листьев, и используют отдельный параметр (а в случае обобщений случайных деревьев – несколько параметров) в каждой области. Если для верного представления выходной функции требуется дерево, имеющее как минимум n листьев, то для построения такого дерева нужно, по крайней мере, n обучающих примеров. А для обеспечения хоть какой-то статистической уверенности в предсказаниях дерева примеров нужно в несколько раз больше n.

В общем случае для различения $O(k)$ областей в пространстве входов всем этим методам нужно $O(k)$ примеров. Как правило, существует $O(k)$ параметров, и с каждой областей ассоциировано $O(1)$ параметров. На рис. 5.10 показан случай использова-

ния одного ближайшего соседа, когда каждый обучающий пример можно использовать для определения самое большее одной области.

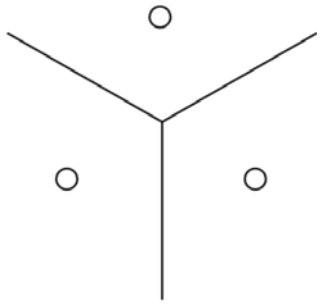

Рис. 5.10 ❖ Как алгоритм ближайшего соседа разбивает пространство входов на области. Пример (представлен кружочком) внутри каждой области определяет ее границы (представлены прямыми линиями). Значение y, ассоциированное с каждым примером, определяет, что будет возвращено для всех точек соответствующей области. Области, определяемые сравнением с ближайшим соседом, образуют геометрическую конструкцию, называемую диаграммой Вороного. Число таких смежных областей не может расти быстрее числа обучающих примеров. Хотя на этом рисунке иллюстрируется поведение конкретного алгоритма ближайшего соседа, другие алгоритмы машинного обучения, опирающиеся для обобщения только на априорное предположение о гладкости, демонстрируют аналогичное поведение: каждый обучающий пример информирует алгоритм обучения только о том, как производить обобщение в некоторой окрестности этого примера.

Существует ли способ представить сложную функцию, для которой число подлежащих различению областей гораздо больше числа обучающих примеров? Очевидно, что одного предположения о гладкости функций для этого недостаточно. Предположим, к примеру, что выходная функция напоминает шахматную доску. Вариативность шахматной доски велика, но все варианты укладываются в простую структуру. Подумаем, что произойдет, если число обучающих примеров намного меньше числа черных и белых клеток. Опираясь только на локальное обобщение и гипотезу о гладкости или локальном постоянстве, можно гарантировать, что алгоритм обучения правильно угадает цвет новой точки, если она находится в одной клетке с каким-нибудь обучающим примером. Но нет никакой гарантии, что алгоритм обучения сумеет правильно обобщить структуру шахматной доски на точки, попадающие в клетки, где нет обучающих примеров. Если у нас нет ничего, кроме этого априорного предположения, то пример может сообщить лишь о цвете своей клетки, и, чтобы узнать цвета всех клеток шахматной доски, мы должны поместить в каждую хотя бы один пример.

Предположение о гладкости и соответствующие ему непараметрические алгоритмы обучения прекрасно работают, если примеров достаточно, чтобы алгоритм увидел верхние точки большинства пиков и нижние точки большинства впадин истинной функции. Вообще говоря, это справедливо, когда обучаемая функция достаточно гладкая и ее изменение сосредоточено в небольшом числе измерений. В пространстве высокой размерности даже очень гладкая функция может изменяться плавно, но по-разному вдоль каждого измерения. Если к тому же функция ведет себя различно

в разных областях, то обучить ее на наборе примеров может оказаться очень трудно. Если функция сложна (т. е. мы хотим различать гораздо больше областей, чем есть примеров), то есть ли надежда на хорошую обобщаемость?

Ответ на оба вопроса – можно ли эффективно представить сложную функцию и может ли оцененная функция хорошо обобщаться на новые данные – положителен. Ключевая идея заключается в том, что очень большое число областей, порядка $O(2^k)$, можно определить с помощью $O(k)$ примеров, если только мы введем некоторые зависимости между областями посредством дополнительных предположений об истинном порождающем распределении. Таким образом, появляется возможность нелокального обобщения (Bengio and Monperrus, 2005; Bengio et al., 2006c). И чтобы воспользоваться ей, во многих алгоритмах глубокого обучения принимаются явные или неявные предположения, действительные для широкого круга задач ИИ. В других подходах к машинному обучению часто делаются еще более сильные, зависящие от задачи допущения. Например, задачу о шахматной доске было бы легко решить, предположив, что выходная функция периодическая. Обычно такие сильные предположения не включаются в нейронные сети, чтобы сохранить возможность обобщения на гораздо более широкий спектр структур. Структура задач ИИ слишком сложна, чтобы ограничиваться простыми, задаваемыми вручную свойствами типа периодичности, поэтому мы хотим, чтобы алгоритмы обучения основывались на более универсальных предположениях. Основная идея глубокого обучения состоит в том, что данные предположительно были порождены *композицией факторов*, или признаков, возможно, организованных иерархически. Есть много других не менее общих предположений, которые позволяют еще повысить качество алгоритмов глубокого обучения. Эти, на первый взгляд, скромные предположения позволяют добиться экспоненциального улучшения связи между числом примеров и числом различимых областей. Мы опишем этот экспоненциальный выигрыш более точно в разделах 6.4.1, 15.4 и 15.5. Экспоненциальный выигрыш от применения глубоких распределенных представлений противостоит экспоненциальным проблемам, вызванным проклятием размерности.

5.11.3. Обучение многообразий

В основе многих идей машинного обучения лежит концепция многообразия.

Многообразие – это связная область. С точки зрения математики, это множество точек, ассоциированных с окрестностью каждой точки. Из любой точки многообразие локально выглядит как евклидово пространство. В повседневной жизни мы воспринимаем поверхность земли как двумерную плоскость, но на самом деле это сферическое многообразие в трехмерном пространстве.

Из идеи окрестности каждой точки вытекает существование преобразований для перемещения из одного места многообразия в другое. В примере земной поверхности как многообразия мы можем пойти на север, юг, восток и запад.

Хотя у термина «многообразие» есть строгое математическое определение, в машинном обучении его используют менее формально для обозначения связного множества точек в пространстве высокой размерности, которое можно хорошо аппроксимировать, вводя в рассмотрение лишь небольшое число степеней свободы, или измерений. Каждое измерение соответствует локальному направлению изменения. На рис. 5.11 показан пример обучающих данных, лежащих в окрестности одномерного многообразия, погруженного в двумерное пространство. В машинном обучении допускаются многообразия, размерность которых различна в разных точках. Так ча-

сто бывает, когда у многообразия есть точки самопересечения. Например, цифра восемь – это одномерное многообразие всюду, кроме точки самопересечения в центре.

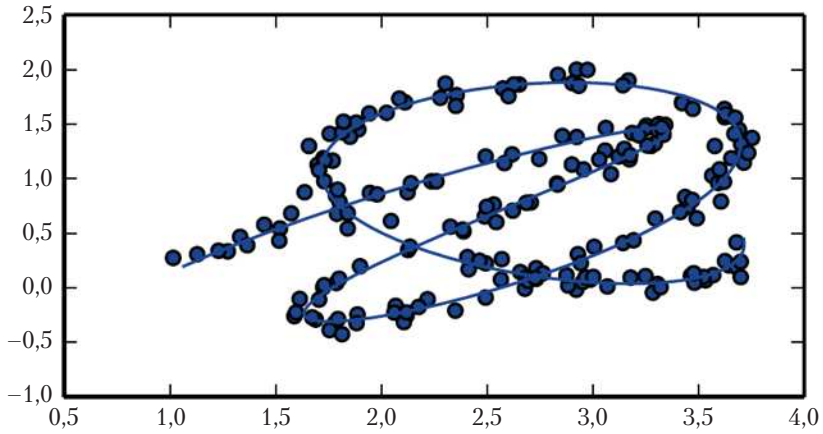

Рис. 5.11 ❖ Выборка данных в двумерном пространстве концентрируется в окрестности одномерного многообразия и напоминает спутанную веревку. Сплошной линией показано многообразие, которое требуется найти в процессе обучения

Многие задачи машинного обучения кажутся безнадежными, если мы ожидаем, что в результате обучения алгоритм должен найти функции с нетривиальными изменениями во всем пространстве $\mathbb{R}^n$. Алгоритмы **обучения многообразий** преодолевают это препятствие, предполагая, что большая часть $\mathbb{R}^n$ – недопустимые входные данные, а интересные входы сосредоточены только в наборе многообразий, содержащем небольшое подмножество точек, причем интересные изменения результирующей обученной функции происходят только вдоль направлений, принадлежащих какому-то одному многообразию, или при переходе с одного многообразия на другое. Обучение многообразий зародилось при рассмотрении непрерывных данных в случае обучения без учителя, хотя сама идея концентрации вероятности обобщается и на дискретные данные, и на обучение с учителем: ключевое допущение заключается в том, что масса вероятности сконцентрирована в малой области.

Предположение о том, что данные расположены вдоль многообразия низкой размерности, не всегда оказывается правильным или полезным. Мы утверждаем, что в задачах ИИ, в частности при обработке изображений, звука или текста, предположение о многообразии, по крайней мере, приближенно правильно. В пользу этой гипотезы можно привести наблюдения двух видов.

Первое наблюдение в пользу **гипотезы о многообразии** заключается в том, что встречающееся в жизни распределение вероятности в изображениях, строках текста и звуковых фрагментах имеет высокую концентрацию. Равномерный шум почти никогда не походит на структурированные входные данные из этих предметных областей. На рис. 5.12 показано, что равномерно распределенные точки выглядят как статический шум на экране аналогового телевизора в отсутствие сигнала. Или, если мы будем генерировать документ, случайным образом выбирая буквы, то какова вероятность получить осмысленный текст на английском языке? Почти нулевая, потому что

большинство длинных последовательностей букв не соответствует последовательностям, встречающимся в естественном языке: распределение последовательностей естественного языка занимает лишь малый объем всего пространства последовательностей букв.

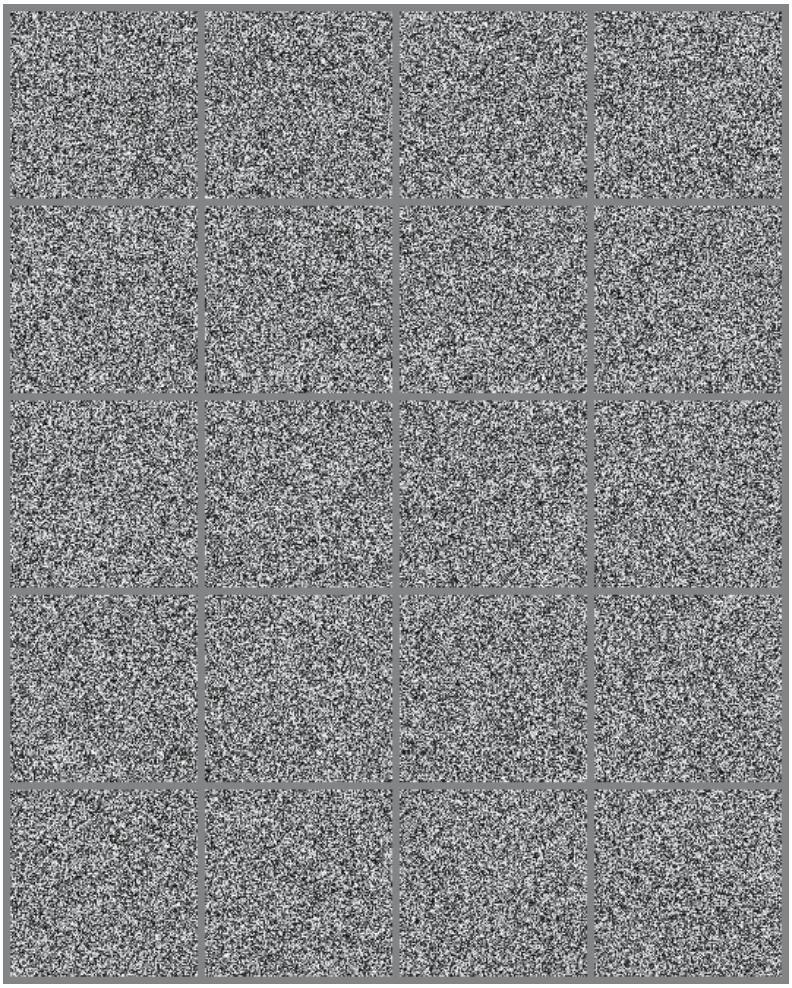

Рис. 5.12 ❖ Результатом случайной выборки пикселей из равномерного распределения является белый шум. Хотя существует ненулевая вероятность сгенерировать таким образом изображение лица или иного объекта, встречающегося в приложениях ИИ, на практике такое не наблюдается. Это означает, что изображения, встречающиеся в приложениях ИИ, занимают пренебрежимо малую долю всего пространства изображений

Конечно, концентрированного распределения вероятности еще недостаточно для доказательства того, что данные расположены на сравнительно небольшом числе многообразий. Нужно еще установить, что предъявленные примеры связаны между собой другими примерами, так что каждый пример окружен похожими на него и до

них можно добраться, применяя преобразования для перемещения по многообразию. Второй аргумент в пользу гипотезы о многообразии состоит в том, что, хотя бы неформально, можно представить себе такие окрестности и такие преобразования. Если речь идет об изображениях, то, безусловно, можно придумать много преобразований, позволяющих перемещаться по многообразию в пространстве изображения: плавное ослабление или усиления освещения, плавный сдвиг или поворот объектов, плавное изменение цвета на поверхностях объектов и т. д. В большинстве приложений, скорее всего, будет присутствовать несколько многообразий. Например, многообразие человеческих лиц вряд ли связано с многообразием кошачьих мордочек.

Эти мысленные эксперименты дают интуитивное обоснование гипотезы о многообразиях. Более строгие эксперименты (Cayton, 2005; Narayanan and Mitter, 2010; Schölkopf et al., 1998; Roweis and Saul, 2000; Tenenbaum et al., 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004) со всей очевидностью поддерживают эту гипотезу для большого класса наборов данных, представляющих интерес для ИИ.

Если данные расположены на многообразии малой размерности, то в алгоритме машинного обучения их наиболее естественно представлять координатами на этом многообразии, а не в $\mathbb{R}^n$. В быту мы рассматриваем дороги как одномерные многообразия, погруженные в трехмерное пространство. Желая сообщить адрес дома, мы указываем его номер относительно дороги, а не координаты в пространстве. Переход в систему координат многообразия – трудная задача, но ее решение обещает заметное улучшение многих алгоритмов машинного обучения. Этот общий принцип применим в самых разных контекстах. На рис. 5.13 показана структура многообразия для набора данных о лицах. К концу книги мы разработаем методы, необходимые для обучения структуре такого многообразия. На рис. 20.6 будет показано, как алгоритм машинного обучения с успехом решает эту задачу.

Рис. 5.13 ❖ Обучающие примеры из набора данных QMUL Multiview Face Dataset (Gong et al., 2000), при составлении которого людей просили позировать так, чтобы покрыть двумерное многообразие, соответствующее двум углам поворота. Мы хотели бы, чтобы алгоритмы обучения смогли выявить координаты на таком многообразии. На рис. 20.6 показан результат решения этой задачи

На этом мы завершаем часть I, где были изложены основы математики и машинного обучения, применяемые в остальных частях книги. Теперь вы готовы приступить к изучению глубокого обучения.

Часть II

ГЛУБОКИЕ СЕТИ: СОВРЕМЕННЫЕ ПОДХОДЫ

В этой части книги описано современное состояние глубокого обучения – как оно применяется для решения практических задач.

У машинного обучения долгая история и многочисленные притязания. Ряд подходов пока не принес желанных плодов. Некоторые амбициозные планы еще ждут своей реализации. Разговор об этих, не столь детально разработанных, разделах глубокого обучения мы отложим до третьей части книги.

А в этой части сосредоточимся на тех подходах, которые доведены до уровня работающих технологий и широко применяются в промышленности.

Современное глубокое обучение предлагает развитую инфраструктуру обучения с учителем. Благодаря добавлению дополнительных слоев и блоков в пределах одного слоя глубокая сеть может представлять все более и более сложные функции. Большинство задач, сводящихся к отображению входного вектора на выходной, с которыми человек справляется легко и непринужденно, может быть решено методами глубокого обучения при наличии достаточно больших моделей и наборов помеченных примеров. Другие задачи, которые нельзя описать как ассоциирование одного вектора с другим, или настолько трудные, что человеку нужно время для их решения, пока не поддаются глубокому обучению.

В этой части книги описаны базовые технологии аппроксимации параметрических функций, лежащие в основе почти всех практических приложений глубокого обучения. Мы начнем с модели глубокой сети прямого распространения, используемой для представления таких функций. Затем мы представим передовые методы регуляризации и оптимизации таких моделей. Для масштабирования моделей на большой объем входных данных, например на изображения высокого разрешения или на длинные временные последовательности, необходима специализация. Мы познакомимся со сверточной сетью, применяемой для масштабирования на большие изображения, и с рекуррентной нейронной сетью для обработки временных последовательностей. Наконец, мы дадим общие рекомендации по практическому проектированию, построению и настройке приложений глубокого обучения и приведем обзор некоторых приложений.

Эти главы наиболее интересны для специалистов-практиков – тех, кто хочет уже сегодня приступить к реализации и использованию алгоритмов глубокого обучения.

Глава 6

Глубокие сети прямого распространения

Глубокие сети прямого распространения, которые называют также **нейронными сетями прямого распространения**, или **многослойными перцептронами** (МСП), – самые типичные примеры моделей глубокого обучения. Цель сети прямого распространения – аппроксимировать некоторую функцию f^*. Например, в случае классификатора $y = f^*(\boldsymbol{x})$ отображает вход $\boldsymbol{x}$ в категорию y. Сеть прямого распространения определяет отображение $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ и путем обучения находит значения параметров $\boldsymbol{\theta}$, дающие наилучшую аппроксимацию.

Слова «**прямое распространение**» означают, что распространение информации начинается с $\boldsymbol{x}$, проходит через промежуточные вычисления, необходимые для определения f, и заканчивается выходом $\boldsymbol{y}$. Не существует обратных связей, по которым выходы модели подаются на ее вход. Обобщенные нейронные сети, включающие такие обратные связи, называются **рекуррентными** и рассматриваются в главе 10.

Сети прямого распространения исключительно важны для практического применения машинного обучения. Они лежат в основе многих важных коммерческих приложений. Например, сверточные сети, используемые для распознавания объектов на фотографиях, – это частный случай сетей прямого распространения. Сети прямого распространения – концептуальная веха на пути к рекуррентным сетям, стоящим за многими приложениями в области естественных языков.

Нейронные сети прямого распространения называются **сетями**, потому что они, как правило, образованы композицией многих различных функций. С моделью ассоциирован ориентированный ациклический граф, описывающий композицию. Например, можно связать три функции $f^{(1)}$, $f^{(2)}$ и $f^{(3)}$ в цепочку $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. Такие цепные структуры чаще всего используются в нейронных сетях. В данном случае $f^{(1)}$ называется **первым слоем** сети, $f^{(2)}$ – **вторым слоем** и т. д. Общая длина цепочки определяет **глубину** модели. Название «глубокое обучение» непосредственно связано с этой терминологией. Последний слой сети прямого распространения называется **выходным**. В ходе обучения нейронной сети мы стремимся приблизить $f(\boldsymbol{x})$ к $f^*(\boldsymbol{x})$. Обучающие данные – это зашумленные приближенные примеры $f^*(\boldsymbol{x})$, вычисленные в различных точках. Каждый пример $\boldsymbol{x}$ сопровождается меткой $y \approx f^*(\boldsymbol{x})$. Обучающие примеры напрямую указывают, что в выходном слое должно соответствовать каждой точке $\boldsymbol{x}$, – это должно быть значение, близкое к y. Поведение остальных слоев напрямую обучающими данными не определяется. Алгоритм обучения должен решить, как использовать эти слои для порождения желаемого выхода, но обучающие дан-

ные ничего не говорят о том, что должен делать каждый слой. Алгоритму обучения предстоит самостоятельно решить, как с помощью этих слоев добиться наилучшей аппроксимации f^*. Поскольку обучающие данные не определяют выходов каждого из этих слоев, они называются **скрытыми слоями**.

Наконец, сети называются *нейронными*, потому что их идея заимствована из нейробиологии. Каждый скрытый слой сети обычно вырабатывает векторные значения. Размерность скрытых слоев определяет **ширину** модели. Каждый элемент вектора можно интерпретировать как нейрон. Вместо того чтобы рассматривать слой как представление функции с векторными аргументами и векторными значениями, мы можем считать, что слой состоит из многих **блоков**, работающих параллельно, и что каждый такой блок представляет функцию, отображающую вектор в скаляр. Каждый блок напоминает нейрон в том смысле, что получает данные от многих других блоков и вычисляет собственное значение активации. Идея использования многих слоев векторных представлений пришла из нейробиологии. Выбор функций $f^{(i)}(\boldsymbol{x})$, используемых для вычисления этих представлений, также изначально навеян экспериментально полученными фактами о функциях, вычисляемых биологическими нейронами. Но в основе современных исследований по нейронным сетям лежат математика и инженерные дисциплины, а перед нейронной сетью не ставится цели точно смоделировать работу мозга. Лучше рассматривать сети прямого распространения не как модели функционирования мозга, а как машины для аппроксимации функций, спроектированные с целью статистического обобщения и иногда использующие наши знания о мозге.

Один из способов разобраться в сетях прямого распространения состоит в том, чтобы начать с линейных моделей и подумать, как преодолеть их ограничения. Линейные модели, в т. ч. логистическая регрессия и линейная регрессия, так привлекательны, потому что дают эффективную и надежную аппроксимацию – в замкнутой форме или посредством выпуклой оптимизации. Но у линейных моделей есть очевидный недостаток – емкость модели ограничена линейными функциями, поэтому модель неспособна понять произвольную связь между двумя величинами.

Чтобы обобщить линейную модель на представление нелинейных функций от $\boldsymbol{x}$, мы можем применить ее не к самому $\boldsymbol{x}$, а к результату вычисления $\boldsymbol{\phi}(\boldsymbol{x})$, где $\boldsymbol{\phi}$ – нелинейное преобразование. Или, что то же самое, применить трюк с ядром, описанный в разделе 5.7.2, для получения нелинейного алгоритма обучения, основанного на неявном применении преобразования $\boldsymbol{\phi}$. Можно считать, что $\boldsymbol{\phi}$ дает набор признаков, описывающих $\boldsymbol{x}$, или новое представление $\boldsymbol{x}$.

Тогда вопрос сводится к выбору отображения $\boldsymbol{\phi}$.

1. Один из вариантов – взять очень общее $\boldsymbol{\phi}$, например бесконечномерное, неявно используемое в ядерных методах, основанных на радиально-базисном ядре. Если размерность $\boldsymbol{\phi}(\boldsymbol{x})$ достаточна велика, то емкости модели хватит для аппроксимации обучающего набора, но обобщаемость на тестовый часто оставляет желать лучшего. Очень общие отображения признаков обычно основаны на принципе локальной гладкости, и закодированной в них априорной информации недостаточно для решения сложных задач.
2. Другой вариант – спроектировать $\boldsymbol{\phi}$ вручную. До наступления эры глубокого обучения так в основном и поступали. Но для каждой задачи требуются десятилетия человеческого труда и специалисты в соответствующей предметной области, например распознавания речи или компьютерного зрения, а передачи знаний между разными областями почти нет.

3. Стратегия глубокого обучения состоит в обучении $\boldsymbol{\phi}$. При таком подходе имеется модель $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$. Теперь у нас есть параметры $\boldsymbol{\theta}$, используемые для обучения $\boldsymbol{\phi}$, выбираемой из широкого класса функций, и параметры $\boldsymbol{w}$, отображающие $\boldsymbol{\phi}(\boldsymbol{x})$ в желаемый выход. Это пример глубокой сети прямого распространения, где $\boldsymbol{\phi}$ определяет скрытый слой. Это единственный из трех подходов, который порывает с предположением о выпуклости задачи обучения, но его достоинства перевешивают недостатки. В этом случае мы параметризуем представление в виде $\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})$ и применяем алгоритм оптимизации для нахождения отображения $\boldsymbol{\phi}$, которому соответствует хорошее представление. Если мы пожелаем, то у этого подхода будут все преимущества общности первого – для этого нужно только взять очень широкое семейство функций $\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})$. Глубокое обучение может также воспользоваться достоинствами второго подхода. Исследователь может включить в модель свои знания, спроектировав семейство функций, которое, по его мнению, должно хорошо обобщаться. Преимущество в том, что человеку нужно только отыскать подходящее семейство функций, а не одну конкретную функцию.

Общий принцип улучшения моделей путем обучения признаков выходит за рамки описываемых в этой главе сетей прямого распространения. Эта тема снова и снова возникает в глубоком обучении и относится ко всем видам моделей, рассматриваемым в книге. Сети прямого распространения – применение этого принципа к обучению детерминированных отображений $\boldsymbol{x}$ в $\boldsymbol{y}$ без обратных связей. Ниже будут представлены другие модели, в которых тот же принцип применяется к обучению стохастических отображений, функций с обратной связью и распределений вероятности одиночного вектора.

Мы начнем эту главу с простого примера сети прямого распространения. Затем мы рассмотрим все проектные решения, принимаемые при развертывании таких сетей. Во-первых, при обучении сети прямого распространения приходится думать о тех же вещах, что для линейных моделей: выборе оптимизатора, функции стоимости и вида выходных блоков. Мы рассмотрим эти основы градиентного обучения, а затем перейдем к решениям, характерным только для сетей прямого распространения. Поскольку в таких сетях есть скрытые слои, то мы должны выбрать **функции активации**, используемые для вычисления вырабатываемых ими значений. Кроме того, нужно спроектировать архитектуру сети: сколько в ней скрытых слоев, как эти слои связаны между собой, сколько блоков в каждом слое. Обучение в контексте глубоких нейронных сетей подразумевает вычисление градиентов сложных функций. Мы опишем алгоритм **обратного распространения** и его современные реализации, позволяющие эффективно вычислять градиенты. И завершим главу историческим обзором.

6.1. Пример: обучение XOR

Чтобы наполнить конкретикой понятие сети прямого распространения, рассмотрим полный пример сети, решающей очень простую задачу: обучение функции XOR.

Функция XOR (исключающее ИЛИ) применяется к двум двоичным значениям, x_1 и x_2. Если ровно одно из них равно 1, то XOR возвращает 1, во всех остальных случаях – 0. Функция XOR является целевой функцией $y = f^*(\boldsymbol{x})$, которую мы хотим обучить. Наша модель описывает функцию $y = f(\boldsymbol{x}; \boldsymbol{\theta})$, а алгоритм обучения должен подобрать параметры $\boldsymbol{\theta}$, так чтобы f была максимально похожа на f^*.

В этом простом примере нас не интересует статистическое обобщение. Мы хотим, чтобы сеть правильно работала на четырех точках $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top, [1, 1]^\top\}$, и будем обучать ее на всех этих точках. Единственная проблема – аппроксимировать обучать набор.

Эту проблему можно рассматривать как задачу регрессии и использовать среднеквадратическую функцию потерь. Мы выбрали такую функцию потерь, чтобы максимально упростить математические выкладки. На практике среднеквадратическая ошибка (СКО) редко подходит для моделирования двоичных данных. Более разумные подходы описаны в разделе 6.2.2.2.

Вычисленная на всем обучающем наборе среднеквадратическая функция потерь имеет вид:

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in\mathbb{X}}(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta}))^2. \tag{6.1}$$

Теперь нужно выбрать форму модели $f(\boldsymbol{x}; \boldsymbol{\theta})$. Допустим, что выбирается линейная модель, в которой $\boldsymbol{\theta}$ состоит из параметров $\boldsymbol{w}$ и b:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top\boldsymbol{w} + b. \tag{6.2}$$

Мы можем минимизировать $J(\boldsymbol{\theta})$ относительно $\boldsymbol{w}$ и b в замкнутой форме с помощью нормальных уравнений.

Решив нормальные уравнения, получаем $\boldsymbol{w} = 0$, $b = 1/2$. Линейная модель просто порождает постоянное выходное значение 0.5. Почему так? По рис. 6.1 видно, что линейная модель не способна представить функцию XOR. Чтобы решить эту проблему, мы можем взять другое пространство признаков, в котором линейной модели будет уже достаточно для представления решения.

Конкретно, рассмотрим простую сеть прямого распространения с одним скрытым слоем, содержащим два скрытых блока. Она показана на рис. 6.2. В этой модели имеется вектор скрытых блоков $\boldsymbol{h}$, вычисляемых функцией $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$. Значения скрытых блоков служат входами для второго слоя, который одновременно является выходным слоем сети. Выходной слой – это просто модель линейной регрессии, но применяется она к $\boldsymbol{h}$, а не к $\boldsymbol{x}$. Теперь сеть содержит две функции, $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ и $y = f^{(2)}(h; \boldsymbol{w}, b)$, а полная модель образована их композицией $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

Что должна вычислять функция $f^{(1)}$? До сих пор линейные модели служили нам верой и правдой, поэтому велико искушение сделать $f^{(1)}$ линейной. К сожалению, если бы $f^{(1)}$ была линейной, то и вся сеть прямого распространения оказалась бы линейной функцией входа. Забудем ненадолго про свободные члены и предположим, что $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top\boldsymbol{x}$, $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top\boldsymbol{w}$. Тогда $f(\boldsymbol{x}) = \boldsymbol{w}^\top\boldsymbol{W}^\top\boldsymbol{x}$. Эту функцию можно представить в виде $f(\boldsymbol{x}) = \boldsymbol{x}^\top\boldsymbol{w}'$, где $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$.

Очевидно, что для описания признаков нужна нелинейная функция. В большинстве нейронных сетей используют композицию аффинного преобразования с обученными параметрами и фиксированной нелинейной функции активации. Воспользуемся этой стратегией и мы, положив $\boldsymbol{h} = g(\boldsymbol{W}^\top\boldsymbol{x} + \boldsymbol{c})$, где $\boldsymbol{W}$ – веса линейного преобразования, а $\boldsymbol{c}$ – смещения. Ранее в модели линейной регрессии мы использовали веса и скалярный параметр смещения для описания аффинного преобразования входного вектора в выходной скаляр. Теперь же мы описываем аффинное преобразование вектора $\boldsymbol{x}$ в вектор $\boldsymbol{h}$, поэтому смещение должно быть вектором. В качестве функции активации обычно берут функцию, применяемую к каждому элементу: $h_i = g(\boldsymbol{x}^\top\boldsymbol{W}_{:,i} + c_i)$. В со-

временных нейронных сетях по умолчанию рекомендуют использовать **блок линейной ректификации** (rectified linear unit – ReLU) (Jarrett et al., 2009; Nair and Hinton, 2010; Glorot et al., 2011a), определяемый функцией активации $g(z) = \max\{0, z\}$, которая изображена на рис. 6.3.

Рис. 6.1 ❖ Решение задачи о функции XOR путем обучения представления. Жирными цифрами обозначены значения, которые обученная функция должна вывести в каждой точке. (*Слева*) Линейная модель, применяемая непосредственно к входным данным, неспособна реализовать функцию XOR. При $x_1 = 0$ выход модели должен возрастать с ростом x_2. При $x_1 = 1$ выход модели должен убывать с ростом x_2. В линейной модели должен быть фиксированный коэффициент между w_2 и x_2. Поэтому линейная модель не может использовать значение x_1 для изменения коэффициента при x_2 и, стало быть, не в состоянии решить задачу. (*Справа*) В преобразованном пространстве признаков, выделяемых нейронной сетью, линейная модель может решить задачу. В нашем случае обе точки, которые должны выводить 1, схлопнуты в одну точку пространства признаков. Иными словами, нелинейное преобразование отображает точки $\boldsymbol{x} = [1, 0]^\top$ и $\boldsymbol{x} = [0, 1]^\top$ в одну точку пространства признаков $\boldsymbol{h} = [1, 0]^\top$. Теперь линейная модель может описать функцию, возрастающую относительно h_1 и убывающую относительно h_2. В этом примере причиной для обучения в пространстве признаков было всего лишь желание увеличить емкость модели, так чтобы она могла аппроксимировать обучающий набор. В реальных приложениях обученное таким образом представление способствует лучшей обобщаемости модели

Теперь можно определить всю сеть целиком:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b. \tag{6.3}$$

Теперь мы можем описать решение задачи о функции XOR. Обозначим

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{6.4}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{6.5}$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{6.6}$$

и положим $b = 0$.

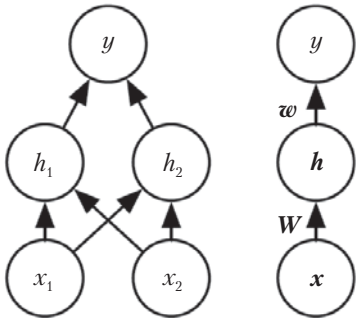

Рис. 6.2 ❖ Пример сети обратного распространения, нарисованной двумя способами. Это та сеть, которой мы воспользовались для решения задачи о функции XOR. В сети имеется скрытый слой с двумя блоками. (*Слева*) Здесь каждый блок представлен вершиной графа. Этот способ явный и однозначный, но когда сеть побольше, чем в этом примере, рисунок занимает слишком много места. (*Справа*) В этом случае вершина графа соответствует целому вектору, представляющему все активации в слое. Такой способ гораздо компактнее. Иногда ребра графа аннотируются именами параметров, описывающих связь между двумя слоями. Здесь мы указали, что матрица $\boldsymbol{W}$ описывает отображение $\boldsymbol{x}$ в $\boldsymbol{h}$, а вектор $\boldsymbol{w}$ – отображение $\boldsymbol{h}$ в y. Как правило, свободные члены для каждого слоя в таких метках опускаются.

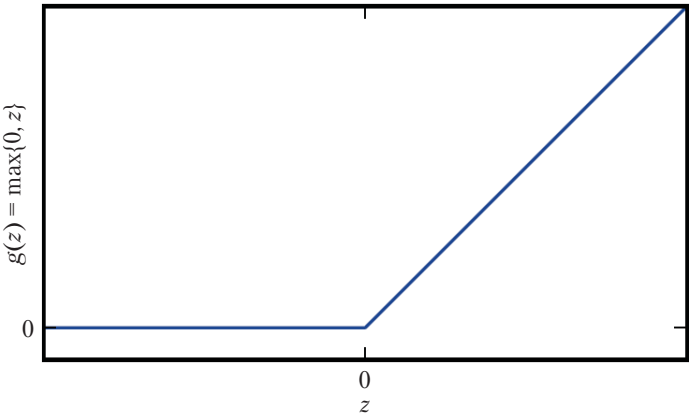

Рис. 6.3 ❖ Ректифицированная линейная функция активации. Эта функция по умолчанию рекомендуется для большинства нейронных сетей прямого распространения. Применение ее к выходу линейного преобразования дает нелинейное преобразование. Функция, впрочем, очень близка к линейной – она является кусочно-линейной с двумя линейными участками. Поскольку блоки линейной ректификации почти линейны, сохраняются многие свойства, благодаря которым линейные модели легко поддаются оптимизации градиентными методами. Сохраняются также свойства, обеспечивающие хорошую обобщаемость линейных моделей. Общий принцип информатики – строить сложные системы из минимальных компонентов. От памяти машины Тьюринга требуется только способность хранить нуль и единицу, а универсальный аппроксиматор можно построить из ректифицированных линейных функций

Посмотрим по шагам, как модель обрабатывает входные данные. Обозначим $\boldsymbol{X}$ матрицу плана, содержащую все четыре точки в пространстве двоичных входов, по одному примеру в строке:

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{6.7}$$

Первый шаг нейронной сети – умножение матрицы входов на матрицу весов первого слоя:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{6.8}$$

Затем она прибавляет вектор смещений $\boldsymbol{c}$, получая в результате

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.9}$$

В этом пространстве все примеры расположены на прямой с угловым коэффициентом 1. При движении вдоль этой прямой выход вначале должен быть равен 0, затем поднимается до 1, потом снова падает до 0. Линейная модель такую функцию реализовать не может. Чтобы завершить вычисление $\boldsymbol{h}$ для каждого примера, применим преобразование линейной ректификации:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.10}$$

Это преобразование изменило соотношение между примерами. Они больше не лежат на одной прямой. Как видно по рис. 6.1, теперь они расположены в пространстве, где линейная модель может решить задачу.

В завершение умножаем на вектор весов $\boldsymbol{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.11}$$

Нейронная сеть получила правильный ответ для каждого входного примера.

В этом примере мы просто задали решение, а затем показали, что его можно получить с нулевой ошибкой. В реальной ситуации количество параметров модели

и обучающих примеров может исчисляться миллиардами, поэтому заранее угадать решение невозможно. Вместо этого применяется алгоритм градиентной оптимизации, который способен найти параметры с очень небольшой ошибкой. Описанное решение задачи о XOR соответствует глобальному минимуму функции потерь, поэтому метод градиентного спуска мог бы сойтись к нему. Существуют эквивалентные решения этой задачи, которые также могли бы быть найдены градиентным спуском. К какой точке сойдется градиентный спуск, зависит от начальных значений параметров. На практике градиентный спуск обычно находит не такие «чистые», целочисленные, с первого взгляда понятные решения, как описанное в этом разделе.

6.2. Обучение градиентными методами

Проектирование и обучение нейронной сети мало отличается от обучения любой другой модели методом градиентного спуска. В разделе 5.10 мы описали, как построить алгоритм машинного обучения путем задания процедуры оптимизации, функции стоимости и семейства моделей.

Основное различие между линейными моделями, встречавшимися нам до сих пор, и нейронными сетями состоит в том, что из-за нелинейности нейронной сети большинство интересных функций потерь оказывается невыпуклыми. Это означает, что нейронные сети обычно обучаются с помощью итеративных градиентных оптимизаторов, которые просто находят очень малое значение функции стоимости, а не с помощью методов решения линейных уравнений, применяемых при обучении моделей регрессии, или алгоритмов выпуклой оптимизации, гарантированно сходящихся к глобальному оптимуму, которые используются при обучении логистической регрессии или модели опорных векторов. Алгоритм выпуклой оптимизации сходится при любых начальных параметрах (теоретически на практике он устойчивый, но может столкнуться с численными проблемами). Метод стохастического градиентного спуска, применяемый к невыпуклым функциям потерь, не дает таких гарантий сходимости и чувствителен к начальным значениям параметров. Для нейронных сетей прямого распространения важно инициализировать все веса небольшими случайными значениями. Смещения можно инициализировать нулями или небольшими положительными значениями. Итеративные алгоритмы градиентной оптимизации, применяемые для обучения сетей прямого распространения и почти всех прочих глубоких моделей, подробно описаны в главе 8, а инициализация параметров обсуждается в разделе 8.4. Пока достаточно понимать, что алгоритм обучения почти всегда основан на использовании градиентного спуска для минимизации функции стоимости тем или иным способом. Конкретные алгоритмы являются улучшением и уточнением идей градиентного спуска, описанных в разделе 4.3, а чаще всего алгоритма стохастического градиентного спуска из раздела 5.9.

Мы, конечно, можем обучать методом градиентного спуска и такие модели, как линейная регрессия и опорные векторы, и так действительно делают, когда обучающий набор очень велик. С этой точки зрения, обучение нейронной сети не сильно отличается от обучения любой другой модели. Для нейронной сети вычисление градиента несколько сложнее, но все равно его можно выполнить эффективно и точно. В разделе 6.5 мы опишем, как найти градиент с помощью алгоритма обратного распространения и его современных модификаций.

Как и для других моделей машинного обучения, для применения обучения на основе градиента нужно выбрать функцию стоимости и способ представления выхода

модели. Давайте вернемся к проектным соображениям, уделив особое внимание случаю нейронных сетей.

6.2.1. Функции стоимости

Важный аспект проектирования глубокой нейронной сети – выбор функции стоимости. По счастью, функции стоимости для нейронных сетей мало отличаются от применяемых в других параметрических моделях, в т. ч. линейных.

В большинстве случаев параметрическая модель определяет распределение $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, и мы должны просто воспользоваться принципом максимального правдоподобия. Это означает, что в роли функции стоимости выступает перекрестная энтропия между обучающими данными и предсказаниями модели.

Иногда принимается более простой подход: вместо предсказания полного распределения вероятности $\boldsymbol{y}$ мы предсказываем какую-то статистику $\boldsymbol{y}$ при условии $\boldsymbol{x}$. Специализированные функции потерь позволяют обучать предиктор таких оценок.

Полная функция стоимости, применяемая при обучении нейронной сети, часто является комбинацией одной из основных функций стоимости, описанных ниже, с регуляризирующим членом. Мы уже встречались с простыми примерами регуляризации в применении к линейным моделям в разделе 5.2.2. Снижение весов, используемое в линейных моделях, безо всяких изменений применимо и к глубоким нейронным сетям и является одной из самых популярных стратегий регуляризации. Более сложные стратегии регуляризации нейронных сетей описаны в главе 7.

6.2.1.1. Обучение условных распределений с помощью максимального правдоподобия

Большинство современных нейронных сетей обучается с помощью максимального правдоподобия. Это означает, что в качестве функции стоимости берется отрицательное логарифмическое правдоподобие, которое можно эквивалентно описать как перекрестную энтропию между обучающими данными и распределением модели. Эта функция задается формулой

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \tag{6.12}$$

Конкретная форма функции стоимости изменяется от модели к модели в зависимости от формы $\log p_{\text{model}}$. Раскрытие этой формулы обычно дает члены, которые не зависят от параметров модели и могут быть отброшены. Например, в разделе 5.5.1 мы видели, что если $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I})$, то можно свести стоимость к среднеквадратической ошибке

$$J(\boldsymbol{\theta}) = {}^1\!/_2 \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \|\boldsymbol{y} - f(\boldsymbol{x}; \boldsymbol{\theta})\|^2 + \text{const} \tag{6.13}$$

с точностью до масштабного коэффициента ${}^1\!/_2$ и члена, не зависящего от $\boldsymbol{\theta}$. Отброшенная константа включает дисперсию нормального распределения, которую мы в данном случае решили не делать параметром. Ранее мы видели, что для линейной модели имеет место эквивалентность между оценкой максимального правдоподобия выходного распределения и минимизацией среднеквадратической ошибки, но на самом деле эта эквивалентность не зависит от функции $f(\boldsymbol{x}; \boldsymbol{\theta})$, используемой для предсказания среднего значения нормального распределения.

Преимущество такого подхода – вывода функции стоимости из оценки максимального правдоподобия – в том, что отпадает необходимость проектировать функцию

стоимости заново для каждой модели. Задание модельного распределения $p(\boldsymbol{y} \mid \boldsymbol{x})$ автоматически определяет функцию стоимости $\log p(\boldsymbol{y} \mid \boldsymbol{x})$.

При проектировании нейронной сети вновь и вновь всплывает одна и та же проблема: если мы хотим использовать градиент функции стоимости для управления алгоритмом обучения, то он должен быть большим и достаточно предсказуемым. Функции с насыщением (становящиеся очень плоскими) не отвечают этому требованию, потому что градиент становится очень малым. Во многих случаях это происходит, потому что функции активации, применяемые для порождения выхода скрытых или выходных блоков, асимптотически горизонтальны. Для многих моделей отрицательное логарифмическое правдоподобие позволяет избежать этой проблемы. Логарифм, входящий в эту функцию стоимости, компенсирует экспоненциальный характер некоторых выходных блоков. Мы обсудим связь между функцией стоимости и выбором выходного блока в разделе 6.2.2.

Одно необычное свойство перекрестной энтропии, используемой при вычислении оценки максимального правдоподобия, заключается в том, что для типичных встречающихся на практике моделей у нее, как правило, нет минимального значения. Если выходная величина дискретна, то в большинстве моделей параметризация устроена так, что модель неспособна представить вероятность 0 или 1, но может подойти к ней сколь угодно близко. Примером может служить логистическая регрессия. В случае вещественных выходных величин, если модель может управлять плотностью выходного распределения (например, путем обучения параметра дисперсии нормального выходного распределения), становится возможным назначение очень высокой плотности правильным выходам обучающего набора, и тогда перекрестная энтропия стремится к отрицательной бесконечности. Методы регуляризации, описанные в главе 7, предлагают еще несколько способов модифицировать задачу обучения, так чтобы модель не смогла таким образом получить неограниченное поощрение.

6.2.1.2. Обучение условных статистик

Вместо обучения полного распределения вероятности $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ мы часто хотим обучить только одну условную статистику $\boldsymbol{y}$ при заданном $\boldsymbol{x}$.

Например, возможно, мы хотим использовать некоторый предиктор $f(\boldsymbol{x}; \boldsymbol{\theta})$ для предсказания среднего значения $\boldsymbol{y}$.

Если нейронная сеть достаточно мощная, то можно считать, что она в состоянии представить любую функцию f из широкого класса функций, связанного только такими условиями, как непрерывность и ограниченность, а не конкретной параметрической формой. В таком случае функцию стоимости можно рассматривать как **функционал**, а не просто функцию. Функционалом называется отображение множества функций во множество вещественных чисел. Процесс обучения можно рассматривать как выбор функции, а не набора параметров. Можно спроектировать функционал стоимости, так чтобы он принимал минимум в некоторой желательной для нас функции, например так чтобы минимум достигался в функции, которая отображает $\boldsymbol{x}$ на ожидаемое значение $\boldsymbol{y}$ при заданном $\boldsymbol{x}$. Для решения задачи оптимизации на множестве функций имеется специальный математический аппарат – **вариационное исчисление**, описываемый в разделе 19.4.2. Для понимания материала этой главы знакомство с вариационным исчислением необязательно. Пока что достаточно понимать, что вариационное исчисление можно использовать для вывода двух сформулированных ниже результатов.

Первый результат заключается в том, что решение задачи оптимизации

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \|\boldsymbol{y} - f(\boldsymbol{x})\|^2 \tag{6.14}$$

имеет вид

$$f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y}\sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}] \tag{6.15}$$

при условии, что эта функция принадлежит классу, по которому мы производим оптимизацию. Иными словами, если бы мы могли обучать на бесконечно большом количестве примеров, выбранных из истинного порождающего распределения данных, то минимизация среднеквадратической ошибки дала бы функцию, которая предсказывает среднее значение $\boldsymbol{y}$ для каждого значения $\boldsymbol{x}$.

Разные функции стоимости дают разные статистики. Второй результат применения вариационного исчисления состоит в том, что уравнение

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \| \boldsymbol{y} - f(\boldsymbol{x})\|_1 \tag{6.16}$$

описывает функцию, которая предсказывает медианное значение $\boldsymbol{y}$ для каждого $\boldsymbol{x}$, при условии что такую функцию можно описать семейством функций, по которому производится оптимизация. Эта функция стоимости обычно называется **средней абсолютной ошибкой**.

К сожалению, среднеквадратическая ошибка и средняя абсолютная ошибка часто дают плохие результаты в сочетании с градиентной оптимизацией. Некоторые выходные блоки с насыщением при использовании таких функций стоимости имеют очень малые градиенты. И это одна из причин, почему перекрестная энтропия в качестве функции стоимости более популярна, чем среднеквадратическая ошибка и средняя абсолютная ошибка, даже в тех случаях, когда нет необходимости оценивать все распределение $p(\boldsymbol{y} \mid \boldsymbol{x})$.

6.2.2. Выходные блоки

Выбор функции стоимости тесно связан с выбором выходного блока. Как правило, мы просто используем перекрестную энтропию между распределением данных и модельным распределением. Выбор представления выхода определяет форму функции перекрестной энтропии.

Любой блок нейронной сети, который можно использовать в качестве выходного, годится и на роль скрытого. Сейчас нас больше интересуют выходные блоки модели, но, в принципе, их можно использовать и во внутренних слоях. Мы еще вернемся к вопросу о скрытых блоках в разделе 6.3.

В этом разделе мы предполагаем, что сеть прямого распространения содержит множество скрытых признаков, описываемое функцией $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$. Тогда роль выходного слоя состоит в том, чтобы предоставить дополнительную информацию, почерпнутую из признаков, для решения задачи, стоящей перед сетью.

6.2.2.1. Линейные блоки для нормальных выходных распределений

Простой выходной блок основан на аффинном преобразовании без нелинейностей. Часто такие блоки называются просто линейными.

Если обозначить признаки $\boldsymbol{h}$, то слой линейных выходных блоков порождает вектор $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$.

Линейные выходные слои часто применяются для порождения среднего условного нормального распределения:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}). \tag{6.17}$$

Тогда максимизация логарифмического правдоподобия эквивалентна минимизации среднеквадратической ошибки.

Схема на основе максимального правдоподобия позволяет также упростить обучение ковариации нормального распределения или сделать ковариацию функцией входа. Однако на ковариационную матрицу необходимо наложить ограничение – она должна быть положительно определенной для всех входов. Удовлетворить такому ограничению с помощью линейного выходного слоя трудно, поэтому обычно для параметризации ковариации используются другие выходные блоки. Подходы к моделированию ковариации кратко описаны в разделе 6.2.2.4.

Поскольку линейные блоки не подвержены насыщению, они не представляют особых проблем для алгоритмов градиентной оптимизации и могут использоваться с самыми разными алгоритмами оптимизации.

6.2.2.2. Сигмоидные блоки и выходное распределение Бернулли

Во многих задачах требуется предсказать значение бинарной величины y. В таком виде можно представить задачу классификации с двумя классами.

Подход на основе максимального правдоподобия заключается в определении распределения Бернулли величины y при условии $\boldsymbol{x}$.

У распределения Бернулли всего один числовой параметр. Нейронная сеть должна предсказать только $P(y = 1 \mid \boldsymbol{x})$. Чтобы эта величина могла интерпретироваться как вероятность, она должна принадлежать отрезку $[0, 1]$.

Для удовлетворения этого условия нужно тщательное проектирование. Допустим, что имеется линейный блок, и обрежем его сверху и снизу, чтобы получить допустимое значение вероятности.

$$P(y = 1 \mid \boldsymbol{x}) = \max\{0, \min\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\}\}. \tag{6.18}$$

Эта формула действительно определяет корректное условное распределение, но эффективно обучить его методом градиентного спуска не получится. Всякий раз, как $\boldsymbol{w}^\top \boldsymbol{h} + b$ выходит за пределы единичного отрезка, градиент выхода модели относительно ее параметров будет равен $\boldsymbol{0}$. Градиент $\boldsymbol{0}$ обычно приводит к проблемам, потому что у алгоритма обучения нет никаких указаний на то, как улучшить параметры.

Лучше применить другой подход, который гарантирует, что градиент обязательно будет достаточно большим, если модель дает неверный ответ. Этот подход основан на использовании сигмоидных выходных блоков в сочетании с максимальным правдоподобием.

Сигмоидный выходной блок определяется формулой

$$\hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{h} + b), \tag{6.19}$$

где σ – логистическая сигмоида, описанная в разделе 3.10.

Можно считать, что сигмоидный выходной блок состоит из двух компонентов. Во-первых, в нем есть линейный слой, вычисляющий $z = \boldsymbol{w}^\top \boldsymbol{h} + b$, а во-вторых, сигмоидная функция активации, преобразующая z в вероятность.

На время забудем о зависимости от $\boldsymbol{x}$ и обсудим, как определить распределение вероятности y, используя значение z. Применение сигмоиды оправдывается возможностью получить ненормированное распределение вероятности $\tilde{P}(y)$, сумма которого не равна 1. Затем мы делим на подходящую константу, чтобы получить правильное распределение вероятности. Если начать с предположения, что логарифм ненормированных вероятностей линейно зависит от y и z, то можно выполнить потенцирование и получить сами ненормированные вероятности. После нормировки получается распределение Бернулли, управляемое сигмоидным преобразованием z:

$$\log\tilde{P}(y) = yz, \tag{6.20}$$

$$\tilde{P}(y) = \exp(yz), \tag{6.21}$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1}\exp(y'z)}, \tag{6.22}$$

$$P(y) = \sigma((2y-1)z). \tag{6.23}$$

Распределения вероятности, основанные на потенцировании и нормировке, часто встречаются в литературе по статистическому моделированию. Переменная z, определяющая такое распределение бинарных величин, называется **логит**.

Этот подход к предсказанию вероятностей в логарифмическом пространстве естественно использовать в сочетании с обучением на основе максимального правдоподобия. Поскольку в этом случае используется функция стоимости $-\log P(y \mid \boldsymbol{x})$, входящий в нее логарифм компенсирует экспоненту в сигмоиде. Не будь этого эффекта, насыщение сигмоиды могло бы помешать градиентному обучению. Функция потерь для обучения параметризованного сигмоидой распределения Бернулли методом максимального правдоподобия имеет вид:

$$J(\boldsymbol{\theta}) = -\log \mathrm{P}(y \mid \boldsymbol{x}) \tag{6.24}$$

$$= -\log \sigma((2y-1)z) \tag{6.25}$$

$$= \zeta((1-2y)z). \tag{6.26}$$

При выводе использованы некоторые свойства из раздела 3.10. Записав потери в терминах функции softplus, мы видим, что насыщение наступает, только когда $(1-2y)z$ принимает большое по абсолютной величине отрицательное значение. Поэтому насыщение имеет место тогда, когда модель уже получила правильный ответ – когда $y = 1$ и z положительно и очень велико или когда $y = 0$ и z отрицательно и очень велико по абсолютной величине. Если знак z не тот, то аргумент функции softplus, $(1-2y)z$ можно упростить до $|z|$. Когда $|z|$ растет при несоответствующем знаке z, функция softplus асимптотически приближается к функции, возвращающей свой аргумент, $|z|$. Производная по z асимптотически приближается к $\mathrm{sign}(z)$, поэтому в пределе – когда z совершенно неправильно – функция softplus вообще не сжимает градиент. Это свойство полезно, потому что означает, что обучение градиентными методами может быстро действовать в направлении быстрого исправления ошибочного z.

При использовании других функций потерь, например среднеквадратической ошибки, потеря может достигать насыщения одновременно с насыщением $\sigma(z)$. Сигмоидная функция активации асимптотически стремится к 0, когда z стремится к минус бесконечности, и к 1, когда z стремится к бесконечности. При таких условиях

сжатие градиента слишком мало и для обучения бесполезно вне зависимости от того, дает модель правильный или неправильный ответ. Поэтому максимальное правдоподобие почти всегда является предпочтительным подходом к обучению сигмоидных выходных блоков.

Аналитически логарифм сигмоиды всегда определен и конечен, поскольку сигмоида возвращает значения из открытого интервала (0, 1), а не из всего замкнутого диапазона допустимых вероятностей [0, 1]. Но в программных реализациях во избежание проблем с численной неустойчивостью лучше записывать отрицательное логарифмическое правдоподобие в виде функции от z, а не от $\hat{y} = \sigma(z)$. Если из-за потери значимости сигмоида обращается в 0, то взятие логарифма $\hat{y}$ дает минус бесконечность.

6.2.2.3. Блоки softmax и категориальное выходное распределение

Если требуется представить распределение вероятности дискретной величины, принимающей n значений, то можно воспользоваться функцией softmax. Ее можно рассматривать как обобщение сигмоиды, которая использовалась для представления распределения бинарной величины.

Функция softmax чаще всего используется как выход классификатора для представления распределения вероятности n классов. Реже функция softmax используется внутри самой модели, когда мы хотим, чтобы модель выбрала один из n вариантов какой-то внутренней переменной.

В случае бинарных величин требовалось породить одно число

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}). \tag{6.27}$$

Поскольку это число должно было находиться между 0 и 1 и поскольку мы хотели, чтобы его логарифм хорошо вел себя при градиентной оптимизации логарифмического правдоподобия, мы решили вместо него порождать число $z = \log\tilde{P}(y = 1 \mid \boldsymbol{x})$. Потенцирование и нормировка дали нам распределение Бернулли, управляемое сигмоидной функцией.

Чтобы обобщить это на случай дискретной случайной величины с n значениями, мы должны породить вектор $\hat{\boldsymbol{y}}$, для которого $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. Мы требуем не только, чтобы каждый элемент $\hat{y}_i$ находился между 0 и 1, но и чтобы сумма всех элементов была равна 1 и таким образом представляла корректное распределение вероятности. Подход, который работает для распределения Бернулли, обобщается и на категориальное распределение. Сначала линейный слой предсказывает ненормированные логарифмы вероятностей:

$$\boldsymbol{z} = \boldsymbol{W}^{\top}\boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

где $z_i = \log\tilde{P}(y = i \mid \boldsymbol{x})$. Функция softmax может затем потенцировать и нормировать $\boldsymbol{z}$ для получения желаемого $\hat{\boldsymbol{y}}$. Формально функция softmax определяется следующим образом:

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_i)}. \tag{6.29}$$

Как и в случае логистической сигмоиды, использование функции exp дает хорошие результаты при обучении softmax с целью порождения выходного значения y

с применением логарифмического правдоподобия. В этом случае мы хотим максимизировать $\log P(\mathrm{y} = i; \boldsymbol{z}) = \log \mathrm{softmax}(\boldsymbol{z})_i$. Определение softmax через exp естественно, потому что логарифм, входящий в логарифмическое правдоподобие, компенсирует потенцирование в softmax:

$$\log \mathrm{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j). \tag{6.30}$$

Первый член в выражении (6.30) показывает, что вход z_i дает прямой вклад в функцию стоимости. Поскольку этот член не испытывает насыщения, то обучение всегда может продолжиться, даже если вклад z_i во второй член становится очень мал. При максимизации логарифмического правдоподобия первый член поощряет увеличение z_i, а второй – уменьшение всех элементов $\boldsymbol{z}$. Чтобы составить интуитивное представление о втором члене $\log \Sigma_j \exp(z_j)$, заметим, что его можно грубо аппроксимировать величиной $\max_j z_j$. В основе такой аппроксимации лежит то соображение, что $\exp(z_k)$ несущественно для любого z_k, значительно меньшего, чем $\max_j z_j$. Отсюда следует, что отрицательное логарифмическое правдоподобие в роли функции стоимости всегда сильнее штрафует самое активное неправильное предсказание. Если правильный ответ уже дает самого большого вклада в softmax, то члены $-z_i$ и $\log \Sigma_j \exp(z_j) \approx \max_j z_j = z_i$ приблизительно взаимно уничтожаются. Такой пример, следовательно, даст малый вклад в общую стоимость обучения, в которой будут преобладать другие примеры, пока еще классифицированные неправильно.

До сих пор мы обсуждали только один пример. В целом нерегуляризованное максимальное правдоподобие побуждает модель обучать параметры, при которых softmax предсказывает долю наблюдений каждого исхода в обучающем наборе:

$$\mathrm{softmax}(\boldsymbol{z}(\boldsymbol{x}; \boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i, \boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}. \tag{6.31}$$

Поскольку максимальное правдоподобие – состоятельная оценка, это гарантированно произойдет при условии, что модельное семейство способно представить обучающее распределение. На практике из-за ограниченной емкости и несовершенной оптимизации модель способна только аппроксимировать эти доли.

Многие целевые функции, отличные от логарифмического правдоподобия, не так хорошо сочетаются с функцией softmax. Конкретно, целевые функции, в которых не используется логарифм для компенсации функции exp, входящей в softmax, не обучаются, когда аргумент exp становится отрицательным и большим по абсолютной величине, что приводит к обнулению градиента. В частности, среднеквадратическая ошибка – плохая функция потерь для блоков softmax, она не всегда побуждает модель изменить свой выход, даже если модель весьма уверенно дает неправильные предсказания (Bridle, 1990). Чтобы понять, в чем тут дело, необходимо внимательно рассмотреть саму функцию softmax.

Как и сигмоида, функция активации softmax склонна к насыщению. У сигмоиды всего один выход, и она насыщается, когда абсолютная величина аргумента очень велика. У softmax выходных значений несколько. Они насыщаются, когда велика абсолютная величина разностей между входными значениями. Когда softmax насыщается, многие основанные на ней функции стоимости также насыщаются, если только они не способны обратить насыщающуюся функцию активации.

Чтобы понять, как softmax реагирует на разность между входными значениями, заметим, что выход softmax инвариантен относительно прибавления одного и того же скаляра ко всем входам:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c). \tag{6.32}$$

Пользуясь этим свойством, мы можем вывести численно устойчивый вариант softmax:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i). \tag{6.33}$$

Этот новый вариант позволяет вычислять softmax с малыми численными погрешностями, даже когда z содержит очень большие по абсолютной величине элементы. Изучив численно устойчивый вариант, мы видим, что на функцию softmax оказывает влияние отклонение ее аргументов от $\max_i z_i$.

Значение $\text{softmax}(\boldsymbol{z})_i$ асимптотически приближается к 1, когда соответствующий аргумент максимален ($z_i = \max_i z_i$) и z_i много меньше остальных входов. Значение $\text{softmax}(\boldsymbol{z})_i$ может также приближаться к 0, когда z_i не максимально, а максимум намного больше. Это обобщение того способа, которым достигается насыщение сигмоидных блоков, и оно может стать причиной аналогичных трудностей при обучении, если функция потерь не компенсирует насыщения.

Аргумент $\boldsymbol{z}$ функции softmax можно породить двумя разными способами. Самый распространенный – просто заставить предыдущий слой нейронной сети выводить каждый элемент $\boldsymbol{z}$, как описано выше на примере линейного слоя $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$. При всей своей простоте этот подход означает, что у распределения избыточное количество параметров. Ограничение, согласно которому сумма n выходов должна быть равна 1, означает, что необходимо только $n - 1$ параметров; вероятность n-го значения можно получить путем вычитания суммы первых $n - 1$ вероятностей из 1. Таким образом, один элемент $\boldsymbol{z}$ можно зафиксировать, например потребовать, чтобы $z_n = 0$. Но это в точности то, что делает сигмоидный блок. Определение $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ эквивалентно определению $P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$ в случае двумерного $\boldsymbol{z}$ и $z_1 = 0$. Оба подхода к softmax – с $n - 1$ и с n аргументами – описывают одно и то же множество распределений вероятности, но обладают разной динамикой обучения. На практике редко возникает существенное различие между перепараметризованным и ограниченным вариантом, а перепараметризованный реализовать проще.

С точки зрения нейробиологии, интересна интерпретация softmax как способа создания некоторой формы конкуренции между блоками, которые в ней участвуют: выходы softmax в сумме всегда дают 1, поэтому увеличение значения одного блока неминуемо влечет уменьшение значений других. Это аналог латерального торможения, которое, как полагают, существует между близкими нейронами коры головного мозга. В крайней своей форме (когда разность между максимальным a_i и всеми остальными велика по абсолютной величине) складывается ситуация, когда **победитель получает все** (один выход почти равен 1, остальные почти равны 0).

Название «softmax» вносит некоторую путаницу. Эта функция ближе к arg max, чем к max. Часть «soft» связана с тем, что функция softmax непрерывная и дифференцируемая. Функция же arg max, результат которой представлен унитарным вектором, не является ни непрерывной, ни дифференцируемой. Следовательно, softmax – это «сглаженный» вариант arg max. Соответствующий сглаженный вариант функции

$\max - \text{softmax}(\boldsymbol{z})^\top \boldsymbol{z}$. Пожалуй, было бы правильнее вместо softmax использовать «soft-argmax», но принятое название уже прочно укоренилось.

6.2.2.4. Другие типы выходных блоков

Описанные выше выходные блоки – линейные, сигмоидные и softmax – применяются наиболее часто. Но нейронные сети обобщаются практически на любой желаемый выходной слой. Принцип максимального правдоподобия подсказывает, как спроектировать хорошую функцию стоимости.

В общем случае, если определено условное распределение $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, то согласно принципу максимального правдоподобия следует использовать в качестве функции стоимости $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$.

Вообще говоря, можно считать, что нейронная сеть представляет функцию $f(\boldsymbol{x}; \boldsymbol{\theta})$. Но значения этой функции не являются прямыми предсказаниями значения y, вместо этого $f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ дает параметры распределения y. Тогда нашу функцию потерь можно интерпретировать как $-\log p(\mathbf{y}; \boldsymbol{\omega}(\boldsymbol{x}))$.

Например, пусть требуется обучить дисперсию условного нормального распределения $\mathbf{y}$ при известном $\mathbf{x}$. В простом случае, когда дисперсия σ^2 постоянна, ее можно выразить в замкнутой форме, потому что оценка максимального правдоподобия дисперсии – это не что иное, как эмпирическое среднее квадратов разностей между наблюдениями и ожидаемыми значениями $\mathbf{y}$. Вычислительно более дорогой подход, не требующий обработки частного случая, состоит в том, чтобы включить дисперсию как одно из свойств распределения $p(\mathbf{y} \mid \boldsymbol{x})$, управляемого функцией $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$. Тогда отрицательное логарифмическое правдоподобие $-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ дает функцию стоимости с подходящими членами, необходимыми, чтобы процедура оптимизации итеративно находила дисперсию. В простом случае, когда стандартное отклонение не зависит от входных данных, мы можем ввести в сеть новый параметр, который непосредственно копируется в $\boldsymbol{\omega}$. Этот новый параметр может совпадать с σ, или называться v и представлять σ^2, или называться β и представлять $1/\sigma^2$ в зависимости от того, как мы хотим параметризовать распределение. Возможно, мы хотим, чтобы модель предсказывала разную величину дисперсии $\mathbf{y}$ для различных значений $\mathbf{x}$. Такая модель называется **гетероскедастической**. В этом случае нужно просто описать дисперсию как одно из значений, порождаемых функцией $f(\mathbf{x}; \boldsymbol{\theta})$. Часто для этой цели выражают нормальное распределение с помощью точности, а не дисперсии, как в уравнении (3.22). В многомерном случае чаще всего используют диагональную матрицу точности

$$\text{diag}(\boldsymbol{\beta}). \tag{6.34}$$

Такое выражение хорошо сочетается с градиентным спуском, потому что формула логарифмического правдоподобия нормального распределения, параметризованного $\boldsymbol{\beta}$, содержит только умножение на β_i и прибавление $\log \boldsymbol{\beta}_i$. Для операций умножения, сложения и логарифмирования градиент ведет себя хорошо. Для сравнения: при параметризации в терминах дисперсии пришлось бы использовать деление. Функция деления становится произвольно крутой в окрестности нуля. Конечно, большие градиенты могут помочь обучению, но если они произвольно велики, то обычно возникает неустойчивость. При параметризации в терминах стандартного отклонения логарифмическое правдоподобие включало бы деление и возведение в квадрат. Последняя операция ведет к обращению градиента в нуль в окрестности нуля, что затрудняет обучение возведенных в квадрат параметров. Но что бы мы ни использова-

ли – стандартное отклонение, дисперсию или точность, необходимо гарантировать, что ковариационная матрица нормального распределения положительно определенная. Поскольку собственные значения матрицы точности обратны собственным значениям ковариационной матрицы, это условие эквивалентно условию положительной определенности матрицы точности. Если использовать диагональную матрицу или единичную матрицу, умноженную на скаляр, то единственное условие, которое необходимо обеспечить для выхода модели, – положительность. Если обозначить $\boldsymbol{a}$ исходную активацию модели, используемой для определения диагональной матрицы точности, то для получения положительного вектора точности можно взять функцию softplus: $\boldsymbol{\beta} = \zeta(\boldsymbol{a})$. Та же стратегия применима при использовании дисперсии или стандартного отклонения вместо точности, или единичной матрицы, умноженной на скаляр, вместо диагональной.

Редко бывает, что ковариационную матрицу или матрицу точности обучают в виде более сложном, чем диагональная матрица. Если ковариация полная и условная, то параметризацию следует выбирать так, чтобы предсказанная ковариационная матрица была положительно определенной. Этого можно добиться, положив $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^{\top}(\boldsymbol{x})$, где $\boldsymbol{B}$ – произвольная квадратная матрица. На практике, правда, возникает проблема: если это матрица полного ранга, то вычисление правдоподобия обходится дорого, т. к. вычисление определителя матрицы $d \times d$ и обратной матрицы $\boldsymbol{\Sigma}^{-1}(\boldsymbol{x})$ (или, что эквивалентно и делается чаще, спектральное разложение $\boldsymbol{\Sigma}(\boldsymbol{x})$ или $\boldsymbol{B}(\boldsymbol{x})$) имеет сложность $O(d^3)$.

Часто мы хотим выполнить многомодальную регрессию, т. е. предсказать вещественные значения, зная условное распределение $p(\boldsymbol{y} \mid \boldsymbol{x})$, которое может иметь несколько пиков в пространстве $\boldsymbol{y}$ для одного и того же значения $\boldsymbol{x}$. В таком случае естественным представлением выхода является гауссова смесь (Jacobs et al., 1991; Bishop, 1994). Нейронные сети с гауссовыми смесями на выходе часто называют **сетями со смесовой плотностью** (mixture density networks). Выход в виде смеси n гауссовых компонент описывается таким условным распределением вероятности:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(c = i \mid \boldsymbol{x}) N(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \Sigma^{(i)}(\boldsymbol{x})). \tag{6.35}$$

Нейронная сеть должна давать три выхода: вектор, определяющий $p(\mathrm{c} = i \mid \boldsymbol{x})$, матрица, задающая $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$ для всех i, и тензор, описывающий $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$ для всех i. Эти выходы должны удовлетворять различным ограничениям.

1. Компоненты смеси $p(\mathrm{c} = i \mid \boldsymbol{x})$: образуют категориальное распределение n компонент, ассоциированное с латентной переменной[1] c, и обычно могут быть получены применением softmax к n-мерному вектору; это гарантирует, что элементы выходного вектора положительны и в сумме дают 1.
2. Средние $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$: задают среднее значение (центр) i-й гауссовой компоненты, никакие ограничения на них не накладываются (обычно в этих выходных блоках нет никакой нелинейности). Если $\mathbf{y}$ – d-мерный вектор, то сеть должна выводить матрицу $n \times d$, содержащую все n этих d-мерных векторов. Обучение средних с применением максимального правдоподобия несколько сложнее, чем обучение

[1] Мы считаем с латентной, потому что не наблюдаем ее в данных: зная вход $\mathbf{x}$ и выход $\mathbf{y}$, невозможно уверенно сказать, какая гауссова компонента отвечает за $\mathbf{y}$, но мы можем предположить, что $\mathbf{y}$ сгенерирована путем выбора одной из них, и сделать этот ненаблюдаемый выбор случайной величиной.

средних распределения с единственной модой. Мы хотим обновлять среднее только для компоненты, которая действительно породила наблюдение. На практике мы не знаем, какая компонента какое наблюдение породила. В выражении отрицательного логарифмического правдоподобия естественно взвешены вклады каждого примера в функцию потерь каждой компоненты, в роли веса выступает вероятность того, что данная компонента породила данный пример.

3. Ковариации $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$: определяют ковариационную матрицу каждой компоненты i. Как и при обучении одной гауссовой компоненты, мы обычно берем диагональную матрицу, чтобы избежать вычисления определителей. Как и при обучении средних смеси, метод максимального правдоподобия осложняется необходимостью приписать часть ответственности за каждую точку компонентам смеси. Алгоритм градиентного спуска автоматически будет следовать за правильным процессом, если ему предоставить корректную спецификацию отрицательного логарифмического правдоподобия для модели смеси.

Сообщалось, что градиентная оптимизация условных гауссовых смесей (на выходе нейронных сетей) может оказаться ненадежной, отчасти из-за операций деления (на дисперсию), которые могут быть численно неустойчивыми (если какая-то дисперсия получается слишком малой для конкретного примера, что приводит к очень большим градиентам). Одно из решений – **обрезать градиенты** (см. раздел 10.11.1), другое – эвристически масштабировать градиенты (Murray and Larochelle, 2014).

Гауссовы смеси на выходе особенно эффективны в порождающих моделях речи (Schuster, 1999) и перемещения физических объектов (Graves, 2013). Стратегия смесовой плотности позволяет сети представить многомодальный выход и управлять дисперсией выхода, что очень важно для получения высококачественного результата в тех предметных областях, где на выходе получаются вещественные числа. Пример сети со смесовой плотностью показан на рис. 6.4.

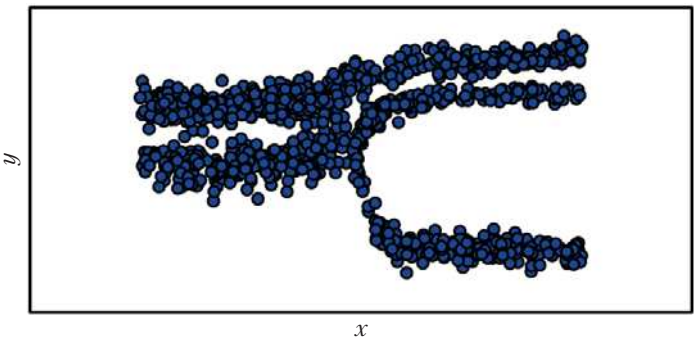

Рис. 6.4 ❖ Примеры получены от нейронной сети с выходным слоем в виде смеси распределений. Вход x выбирается из равномерного распределения, а выход y – из $p_{\text{model}}(y \mid x)$. Нейронная сеть способна обучить нелинейные отображения входа на параметры выходного распределения. В состав этих параметров входят вероятности, управляющие тем, какая из трех компонент смеси порождает выход, а также параметры отдельных компонент. Каждая компонента смеси – нормальное распределение с предсказанными средним и дисперсией. Все эти аспекты выходного распределения могут изменяться в зависимости от x, причем нелинейно

В общем случае мы можем захотеть продолжить моделирование векторов $\boldsymbol{y}$, содержащих все больше переменных, и налагать на эти переменные все более сложную структуру. Например, если мы хотим, чтобы нейронная сеть выводила последовательность символов, образующих предложение, то можем и дальше применять принцип максимального правдоподобия к нашей модели $p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$. В этом случае модель, используемая для описания $\boldsymbol{y}$, становится слишком сложной для рассмотрения в этой главе. В главе 10 мы опишем, как использовать рекуррентные нейронные сети для описания таких моделей последовательностей, а в части III опишем передовые методы моделирования произвольных распределений вероятности.

6.3. Скрытые блоки

До сих пор мы ограничивались обсуждением проектных решений при построении нейронных сетей, типичных для большинства параметрической моделей машинного обучения, обучаемых методами градиентной оптимизации. Теперь займемся проблемой, возникающей только в нейронных сетях прямого распространения: как выбрать тип блока в скрытых слоях модели.

Проектирование скрытых блоков – чрезвычайно активная область исследований, в которой не так уж много руководящих теоретических принципов.

Блоки линейной ректификации – отличный выбор для скрытых блоков в отсутствие дополнительных аргументов. Но есть и много других типов. Бывает трудно решить, какой тип взять в конкретном случае (хотя обычно блоки линейной ректификации дают приемлемый результат). Мы опишем кое-какие интуитивные соображения, стоящие за выбором типа скрытого блока. Они помогут принять решение, но заранее предсказать, какой тип окажется оптимальным, как правило, невозможно. Процесс проектирования – это последовательность проб и ошибок, когда высказывается гипотеза о подходящем блоке, затем обучается сеть с таким типом скрытых блоков и результат оценивается на контрольном наборе.

Некоторые скрытые блоки, включенные в список, не являются всюду дифференцируемыми. Например, функция линейной ректификации $g(z) = \max\{0, z\}$ не дифференцируема в точке $z = 0$. Может показаться, что из-за этого g непригодна для работы с алгоритмом обучения градиентными методами. Но на практике градиентный спуск работает для таких моделей машинного обучения достаточно хорошо. Отчасти это связано с тем, что алгоритмы обучения нейронных сетей обычно не достигают локального минимума функции стоимости, а просто находят достаточно малое значение, как показано на рис. 4.3 (эти идеи найдут дальнейшее развитие в главе 8). Поскольку мы не ожидаем, что обучение выйдет на точку, где градиент равен 0, то можно смириться с тем, что минимум функции стоимости соответствует точкам, в которых градиент не определен. Недифференцируемые скрытые блоки обычно не дифференцируемы лишь в немногих точках. В общем случае функция $g(z)$ имеет производную слева, определяемую коэффициентом наклона функции слева от z, и аналогично производную справа. Функция дифференцируема в точке z, только если производные слева и справа определены и равны между собой. Для функций, встречающихся в контексте нейронных сетей, обычно определены производные слева и справа. Для функции $g(z) = \max\{0, z\}$ производная слева в точке $z = 0$ равна 0, а производная справа – 1. В программных реализациях обучения нейронной сети обычно возвращается какая-то односторонняя производная, а не сообщается, что производная не определена и не возбуждается исключение. Эври-

стически это можно оправдать, заметив, что градиентная оптимизация на цифровом компьютере в любом случае подвержена численным погрешностям. Когда мы просим вычислить $g(0)$, крайне маловероятно, что истинное значение действительно равно 0. Скорее всего, это какое-то малое значение ε, округленное до 0. В некоторых случаях возможны теоретически более убедительные обоснования, но обычно к обучению нейронных сетей они не относятся. Важно, что на практике можно спокойно игнорировать недифференцируемость функций активации скрытых блоков, описанных ниже.

В большинстве случаев скрытый блок можно описать следующим образом: получить вектор входов $\boldsymbol{x}$, вычислить аффинное преобразование $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$ и применить к каждому элементу нелинейную функцию $g(\boldsymbol{z})$. Друг от друга скрытые блоки отличаются только функцией активации $g(\boldsymbol{z})$.

6.3.1. Блоки линейной ректификации и их обобщения

В блоке линейной ректификации используется функция активации $g(z) = \max\{0, z\}$.

Эти блоки легко оптимизировать, потому что они очень похожи на линейные. Разница только в том, что блок линейной ректификации в половине своей области определения выводит 0. Поэтому производная блока линейной ректификации остается большой всюду, где блок активен. Градиенты не только велики, но еще и согласованы. Вторая производная операции ректификации всюду равна нулю, а первая производная равна 1 всюду, где блок активен. Это означает, что направление градиента гораздо полезнее для обучения, чем в случае, когда функция активации подвержена эффектам второго порядка.

Блоки линейной ректификации обычно применяются после аффинного преобразования:

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}). \tag{6.36}$$

При инициализации параметров аффинного преобразования рекомендуется присваивать всем элементам $\boldsymbol{b}$ небольшое положительное значение, например 0.1. Тогда блок линейной ректификации в начальный момент с большой вероятностью окажется активен для большинства обучающих примеров, и производная будет отлична от нуля.

Существует несколько обобщений блока линейной ректификации. Большинство из них демонстрирует более высокое качество лишь в отдельных случаях.

Недостатком блоков линейной ректификации является невозможность обучить их градиентными методами на примерах, для которых функция активации блока равна нулю. Различные обобщения гарантируют, что градиент имеется в любой точке.

Три обобщения блоков линейной ректификации основаны на использовании ненулевого углового коэффициента α_i, когда $z_i < 0$: $h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$. В случае абсолютной ректификации (absolute value rectification) берутся фиксированные значения $\alpha_i = -1$, так что $g(z) = |z|$. Такая функция активации используется при распознавании объектов в изображении (Jarrett et al., 2009), где имеет смысл искать признаки, инвариантные относительно изменения полярности освещения. Другие обобщения находят более широкие применения. В случае **ReLU с утечкой** (leaky ReLU) (Maas et al., 2013) α_i принимаются равными фиксированному малому значению, например 0.01, а в случае **параметрического ReLU**, или **PReLU** α_i, считается обучаемым параметром (He et al., 2015).

Maxout-блоки (Goodfellow et al., 2013a) – это дальнейшее обобщение блоков линейной ректификации. Вместо того чтобы применять функцию $g(z)$ к каждому эле-

менту, вектор $\boldsymbol{z}$ разбивается на группы по k значений. Затем каждый maxout-блок выводит максимальный элемент одной из групп:

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j, \tag{6.37}$$

где $\mathbb{G}^{(i)}$ – множество индексов входов, входящих в i-ю группу, $\{(i-1)k+1, ..., ik\}$. Это позволяет обучать кусочно-линейную функцию, дающую отклик в нескольких направлениях в пространстве входов $\boldsymbol{x}$.

Maxout-блок может обучить кусочно-линейную выпуклую функцию, состоящую из k участков. Поэтому такие блоки можно рассматривать как средство *обучения самой функции активации*, а не просто связи между блоками. При достаточно больших k maxout-блок может научиться аппроксимировать любую выпуклую функцию с произвольной точностью. В частности, maxout-слой с двумя участками линейности можно обучить реализации той же функции от входа $\boldsymbol{x}$, что и традиционный слой с блоком линейной ректификации, абсолютной ректификации, ректификации с утечкой или параметрической ректификации, а также реализации совершенно другой функции. Разумеется, maxout-слой параметризуется не так, как слои других типов, поэтому динамика обучения будет иной даже в случае, когда maxout обучают реализации той же функции от $\boldsymbol{x}$, что и другие слои.

Каждый maxout-блок параметризуется k векторами весов вместо одного, поэтому для них нужно больше регуляризации, чем для блоков линейной ректификации. Они могут хорошо работать вообще без регуляризации, если обучающий набор достаточно велик, а количество участков линейности в каждом блоке мало (Cai et al., 2013).

У maxout-блоков есть еще несколько преимуществ. В некоторых случаях можно получить статистический и вычислительный выигрыш от уменьшения числа параметров. Точнее, если признаки, собранные n разными линейными фильтрами, можно обобщить без потери информации, взяв максимум по каждой группе k признаков, то следующий уровень может обойтись в k раз меньшим числом весов.

Поскольку каждый блок «питается» несколькими фильтрами, maxout-блоки обладают некоторой избыточностью, позволяющей им противостоять феномену **катастрофической забывчивости**, когда нейронная сеть забывает, как решать задачи, которым ее обучили в прошлом (Goodfellow et al., 2014a).

Блоки линейной ректификации и все их обобщения основаны на принципе, согласно которому модель проще обучить, если ее поведение близко к линейному. Тот же общий принцип использования линейного поведения для упрощения оптимизации применим не только к глубоким линейным сетям. Рекуррентные сети могут обучаться на последовательностях и порождать последовательность состояний и выходов. В ходе их обучения необходимо передавать информацию от одного шага к другому, что гораздо проще, когда производятся линейные вычисления (некоторые производные по направлению близки к 1). В одной из самых лучших архитектур рекуррентных сетей, LSTM, информация распространяется во времени путем суммирования – особенно простого вида линейной активации. Мы вернемся к этой теме в разделе 10.10.

6.3.2. Логистическая сигмоида и гиперболический тангенс

Блоки линейной ректификации стали использоваться сравнительно недавно, а раньше в большинстве нейронных сетей в роли функции активации применялась логистическая сигмоида

$$g(z) = \sigma(z) \tag{6.38}$$

или гиперболический тангенс

$$g(z) = \tanh(z). \tag{6.39}$$

Эти функции активации тесно связаны: $\tanh(z) = 2\sigma(2z) - 1$.

Мы уже видели сигмоидальные блоки в качестве выходных, предсказывающих вероятность того, что бинарная величина равна 1. В отличие от кусочно-линейных, сигмоидальные блоки близки к асимптоте в большей части своей области определения – приближаются к высокому значению, когда z стремится к бесконечности, и к низкому, когда z стремится к минус бесконечности. Высокой чувствительностью они обладают только в окрестности нуля. Из-за насыщения сигмоидальных блоков градиентное обучение сильно затруднено. Поэтому использование их в качестве скрытых блоков в сетях прямого распространения ныне не рекомендуется. Применение же в качестве выходных блоков совместимо с обучением градиентными методами, если функция стоимости компенсируется насыщением сигмоиды в выходном слое.

Если использовать сигмоидальную функцию активации необходимо, то лучше взять не логистическую сигмоиду, а гиперболический тангенс. Он ближе к тождественной функции в том смысле, что $\tanh(0) = 0$, тогда как $\sigma(0) = 1/2$. Поскольку tanh походит на тождественную функцию в окрестности нуля, обучение глубокой нейронной сети $\hat{y} = \boldsymbol{w}^\top \tanh(\boldsymbol{U}^\top \tanh(\boldsymbol{V}^\top \boldsymbol{x}))$ напоминает обучение линейной модели $\hat{y} = \boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$, при условии что сигналы активации сети удается удерживать на низком уровне. При этом обучение сети с функцией активации tanh упрощается.

Сигмоидальные функции активации все же применяются, но не в сетях прямого распространения. К рекуррентным сетям, многим вероятностным моделям и некоторым автокодировщикам предъявляются дополнительные требования, исключающие использование кусочно-линейных функций активации и делающие сигмоидальные блоки более подходящими, несмотря на проблемы насыщения.

6.3.3. Другие скрытые блоки

Существует много других типов скрытых блоков, но используются они реже.

Вообще говоря, многие дифференцируемые функции показывают отличные результаты. Есть целый ряд неопубликованных функций активации, которые ведут себя ничуть не хуже популярных. Приведем конкретный пример: мы тестировали сеть прямого распространения с функцией $\boldsymbol{h} = \cos(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ на наборе данных MNIST и получили частоту ошибок менее 1 процента, что сравнимо с результатами, полученными с использованием более традиционных функций активации. В ходе исследований и разработки новых методов нередко тестируется много разных функций активации и обнаруживается, что результаты, полученные при отходе от стандартной практики, вполне сопоставимы. Это означает, что новые типы скрытых блоков обычно публикуются только в случае, когда улучшение весомо и очевидно. Скрытые блоки, работающие примерно так же, как известные, – дело настолько обычное, что рассматривать их неинтересно.

Бессмысленно перечислять все типы скрытых блоков, описанные в литературе. Мы отметим только несколько особенно полезных и непохожих на другие.

Одна из возможностей – не использовать функцию активации $g(z)$ вовсе. Можно считать, что в этой роли выступает тождественная функция. Мы уже видели, что

линейный блок может быть полезен в выходном слое нейронной сети. Его можно использовать и в качестве скрытого блока. Если каждый слой сети состоит только из линейных преобразований, то сеть в целом будет линейной. Однако некоторые слои могут быть и чисто линейными – это вполне нормально. Рассмотрим слой нейронной сети, имеющий n входов и p выходов, $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$. Его можно заменить двумя слоями, в одном из которых используется матрица весов $\boldsymbol{U}$, а в другом – матрица весов $\boldsymbol{V}$. Если в первом слое нет функции активации, то мы, по сути дела, разложили на множители матрицу весов исходного слоя, основанного на $\boldsymbol{W}$. Подход, основанный на разложении в произведение, состоит в том, чтобы вычислить $\boldsymbol{h} = g(\boldsymbol{V}^\top \boldsymbol{U}^\top \boldsymbol{x} + \boldsymbol{b})$. Если $\boldsymbol{U}$ порождает q выходов, то $\boldsymbol{U}$ и $\boldsymbol{V}$ вместе содержат только $(n + p)q$ параметров, тогда как $\boldsymbol{W}$ – np параметров. Для малых q экономия параметров может быть существенной. Платой за это является ограничение – линейное преобразование должно иметь низкий ранг, но таких низкоранговых связей часто достаточно. Таким образом, линейные скрытые блоки предлагают эффективный способ уменьшить число параметров сети.

Блоки softmax – еще один вид блоков, часто используемых на выходе (см. раздел 6.2.2.3), но иногда они применяются и в скрытых слоях. Такие блоки естественно представляют распределение вероятности дискретной переменной, принимающей k значений, поэтому они могут играть роль своего рода переключателей. Обычно такие скрытые блоки применяются только в более сложных архитектурах, которые явно обучаются манипулировать памятью (см. раздел 10.12).

Из других довольно распространенных скрытых блоков упомянем следующие:

- блок в виде **радиально-базисной функции** (RBF): $h_i = \exp((-1/\sigma_i^2)\|\boldsymbol{W}_{:,i} - \boldsymbol{x}\|^2)$. Эта функция становится более активной, когда $\boldsymbol{x}$ стремится к образцу $\boldsymbol{W}_{:,i}$. Поскольку для большинства $\boldsymbol{x}$ она асимптотически равна 0, оптимизировать ее трудно;
- **Softplus:** $g(a) = \zeta(a) = \log(1 + e^a)$. Это сглаженный вариант ректификатора, использованный в работе Dugas et al. (2001) для аппроксимации функций и в работе Nair and Hinton (2010) для условных распределений в неориентированных вероятностных моделях. В работе Glorot et al. (2011a) сравнивались softplus и ректификатор и обнаружилось, что результаты последнего лучше. Применение softplus, вообще говоря, не рекомендуется. Функция softplus демонстрирует, что качество скрытых блоков может противоречить интуиции, – казалось бы, она должна иметь преимущество над ректификатором в силу дифференцируемости всюду или не столь полного насыщения, но опыт показывает, что это не так;
- **Hardtanh**. По форме эта функция похожа на tanh и на ректификатор, но, в отличие от последнего, ограничена: $g(a) = \max(-1, \min(1, a))$. Она введена в работе Collobert (2004).

В области проектирования скрытых слоев до сих пор ведутся активные исследования, и многие полезные типы слоев ждут своего открытия.

6.4. Проектирование архитектуры

Еще один ключевой аспект нейронных сетей – задание архитектуры. Под **архитектурой** понимается общая структура сети: сколько в ней должно быть блоков и как эти блоки соединены между собой.

Большинство нейронных сетей организовано в виде групп блоков, именуемых слоями. В большинстве архитектур нейронных сетей слои собраны в цепочку, так что каждый слой является функцией от предыдущего. В этой структуре первый слой задается уравнением

$$\boldsymbol{h}^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}), \tag{6.40}$$

второй слой – уравнением

$$\boldsymbol{h}^{(2)} = g^{(2)}(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \tag{6.41}$$

и так далее.

В таких цепных архитектурах главное – выбор глубины сети и ширины каждого слоя. Как мы увидим, даже сети всего с одним скрытым слоем достаточно для аппроксимации обучающего набора. Более глубокие сети часто способны обходиться гораздо меньшим числом блоков в одном слое и гораздо меньшим числом параметров и, кроме того, хорошо обобщаются на тестовый набор, но их труднее оптимизировать. Идеальная архитектура сети для данной задачи ищется в ходе экспериментов, направление которых определяется ошибкой на контрольном наборе.

6.4.1. Свойства универсальной аппроксимации и глубина

Линейная модель, отображающая признаки на выходы посредством умножения на матрицу, по определению способна представить только линейные функции. Ее преимущество – простота обучения, потому что многие функции потерь в случае применения к линейной модели приводят к задачам выпуклой оптимизации. К сожалению, от систем часто требуется обучать нелинейные функции.

На первый взгляд кажется, что для обучения нелинейной функции нужно проектировать специализированное семейство моделей для каждого вида нелинейности. По счастью, сети прямого распространения со скрытыми слоями предоставляют универсальную инфраструктуру аппроксимации. Точнее, **универсальная теорема аппроксимации** (Hornik et al., 1989; Cybenko, 1989) утверждает, что сеть прямого распространения с линейным выходным слоем и, по крайней мере, одним скрытым слоем с произвольной «сплющивающей» функцией активации (такой, например, как логистическая сигмоида) может аппроксимировать любую измеримую по Борелю функцию, отображающую одно конечномерное пространство в другое с любой точностью, при условии что в сети достаточно скрытых блоков. Производные сети прямого распространения могут также аппроксимировать производные функции с любой точностью (Hornik et al., 1990). Понятие измеримости по Борелю выходит за рамки этой книги; нам достаточно знать, что любая непрерывная функция на замкнутом ограниченном подмножестве $\mathbb{R}^n$ измерима по Борелю и потому может быть аппроксимирована нейронной сетью. Нейронная сеть также может аппроксимировать произвольную функцию, отображающую одно конечномерное дискретное пространство в другое. Хотя первоначально эти теоремы были доказаны в терминах блоков с функциями активации, которые насыщаются при стремлении аргумента к бесконечности любого знака, универсальные теоремы аппроксимации справедливы также для более широкого класса функций активации, включающего ныне широко используемые блоки линейной ректификации (Leshno et al., 1993).

Универсальная теорема аппроксимации означает, что какой бы ни была обучаемая функция, найдется достаточно большой МСП, способный ее *представить*. Однако не

гарантируется, что алгоритм обучения сможет эту функцию *обучить*. Даже если МСП может представить функцию, обучение может оказаться невозможным по двум причинам. Во-первых, применяемый при обучении алгоритм оптимизации может не найти соответствующие ей значения параметров. Во-вторых, из-за переобучения алгоритм оптимизации может выбрать не ту функцию. Напомним (раздел 5.2.1), что согласно теореме об отсутствии бесплатных завтраков не существует универсального верховного алгоритма машинного обучения. Сети прямого распространения являются универсальной системой представления функций в том смысле, что для любой заданной функции найдется аппроксимирующая ее сеть. Но не существует универсальной процедуры исследования набора конкретных обучающих примеров и выбора функции, которая хорошо обобщалась бы на примеры, не принадлежащие этому набору.

По универсальной теореме аппроксимации, существует достаточно большая сеть, аппроксимирующая функцию с любой точностью, но теорема ничего не говорит о том, насколько велика эта сеть. В работе Barron (1993) даны верхние границы размера сети с одним уровнем, необходимой для аппроксимации широкого класса функций. К сожалению, в худшем случае может понадобиться экспоненциальное число скрытых блоков (возможно, по одному скрытому блоку на каждую нуждающуюся в различении конфигурацию входов). Проще всего убедиться в этом в бинарном случае: число возможных бинарных функций от векторов $\boldsymbol{v} \in \{0, 1\}^n$ равно 2^{2^n}, и для выбора одной такой функции нужно 2^n бит, так что в общем случае необходимо $O(2^n)$ степеней свободы.

Короче говоря, сети прямого распространения с одним слоем достаточно для представления любой функции, но этот слой может оказаться невообразимо большим, не поддающимся обучению и плохо обобщающимся. Во многих случаях применение глубоких моделей позволяет уменьшить число необходимых блоков и величину ошибки обобщения.

Различные семейства функций можно эффективно аппроксимировать с помощью архитектуры с глубиной, большей некоторой величины d, но требуется гораздо большая по размеру модель, если глубина должна быть меньше или равна d. Во многих случаях количество скрытых блоков, необходимое «мелкой» модели, экспоненциально зависит от n. Подобные результаты сначала были доказаны для моделей, ничем не напоминающих непрерывные дифференцируемые нейронные сети, применяемые в машинном обучении, но затем обобщены и на такие модели. Первые результаты были получены для электрических схем, состоящих из логических вентилей (Håstad, 1986). Впоследствии они были обобщены на линейные пороговые блоки с неотрицательными весами (Håstad and Goldmann, 1991; Hajnal et al., 1993), а еще позже – на сети с непрерывными функциями активации (Maass, 1992; Maass et al., 1994). Во многих современных нейронных сетях используются блоки линейной ректификации. В работе Leshno et al. (1993) показано, что мелкие сети с широким классом неполиномиальных функций активации, включающим и блоки линейной ректификации, обладают свойствами универсальной аппроксимации, но эти результаты не затрагивают вопроса о глубине или эффективности – они утверждают лишь, что с помощью достаточно большой сети ректификаторов можно представить произвольную функцию. В работе Montufar et al. (2014) показано, что для функций, представимых глубокой сетью ректификаторов, может потребоваться экспоненциальное число скрытых блоков, если сеть мелкая (один скрытый слой). Точнее, показано, что кусочно-линейные сети (получаемые с помощью нелинейностей типа ректификаторов или maxout-блоков) могут

представить любые функции, но число линейных участков экспоненциально зависит от глубины сети. На рис. 6.5 показано, как сеть с абсолютной ректификацией создает зеркальное изображение функции, вычисленной некоторым скрытым блоком, относительно входа этого блока. Каждый скрытый блок задает место, где нужно «перегнуть» пространство входов, чтобы получить зеркальный отклик (по обе стороны абсолютной нелинейности). С помощью композиции таких операций перегибания мы получаем экспоненциально растущее число участков нелинейности и тем самым можем уловить любые регулярные (т. е. повторяющиеся) паттерны.

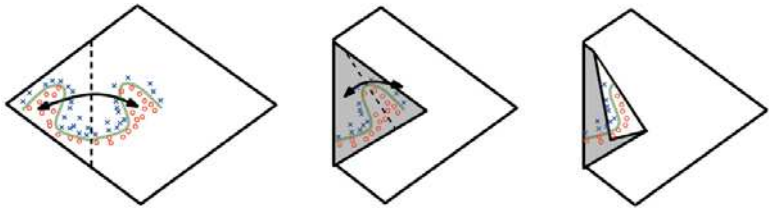

Рис. 6.5 ❖ Интуитивное геометрическое объяснение экспоненциального характера глубоких сетей ректификаторов, формально доказанного в работе Montufar et al. (2014). (*Слева*) Блок абсолютной ректификации порождает одинаковые выходы для любой пары зеркально симметричных входов. Ось симметрии задается гиперплоскостью, определяемой весами и смещением блока. Функция, вычисляемая этим блоком (зеленая решающая поверхность), будет зеркальным отражением более простого паттерна относительно этой оси симметрии. (*В центре*) Функцию можно получить путем перегибания пространства по этой оси симметрии. (*Справа*) На первый повторяющийся паттерн можно наложить еще один (с помощью блока следующего слоя) и получить тем самым еще одну симметрию (четырехкратное повторение при двух скрытых слоях). Рисунок взят из работы Montufar et al. (2014) с разрешения авторов

Основная теорема в работе Montufar et al. (2014) утверждает, что число линейных участков, представимых глубокой сетью ректификаторов с d входами, глубиной l и n блоками в каждом скрытом слое, равно

$$O\left(\left(\frac{n}{d}\right)^{d(l-1)} n^d\right), \tag{6.42}$$

т. е. экспоненциально зависит от глубины l. В случае maxout-сетей с k фильтрами на один блок число линейных участков равно

$$O(k^{(l-1)+d}). \tag{6.43}$$

Конечно, нет никакой гарантии, что функции, которые мы хотим обучать в приложениях машинного обучения (и особенно ИИ), обладают таким свойством.

Выбор глубокой модели может быть продиктован и статистическими соображениями. При выборе конкретного алгоритма машинного обучения мы неявно формулируем некоторые априорные предположения о характере функции, которую этот алгоритм должен обучить. В глубокой модели закодирована очень общая гипотеза, согласно которой обучаемая функция должна быть композицией более простых.

С точки зрения обучения представлений, эту гипотезу можно интерпретировать, сказав, что задача обучения состоит в выявлении множества истинных факторов вариативности, которое, в свою очередь, можно описать в терминах других, более простых факторов вариативности. По-другому использование глубокой архитектуры можно интерпретировать как выражение нашей веры в том, что функция, которую мы хотим обучить, является компьютерной программой, состоящей из нескольких шагов, на каждом из которых используется результат предыдущего шага. Эти промежуточные результаты не обязательно являются факторами вариативности, их можно уподобить счетчикам или указателям, с помощью которых сеть организует свою внутреннюю работу. Эмпирически показано, что для широкого класса задач увеличение глубины влечет улучшение обобщаемости (Bengio et al., 2007; Erhan et al., 2009; Bengio, 2009; Mesnil et al., 2011; Ciresan et al., 2012; Krizhevsky et al., 2012; Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013; Kahou et al., 2013; Goodfellow et al., 2014d; Szegedy et al., 2014a). Примеры таких эмпирических результатов приведены на рис. 6.6 и 6.7. Они наводят на мысль, что глубокие архитектуры действительно выражают полезную априорную информацию о пространстве функций, обучаемых моделью.

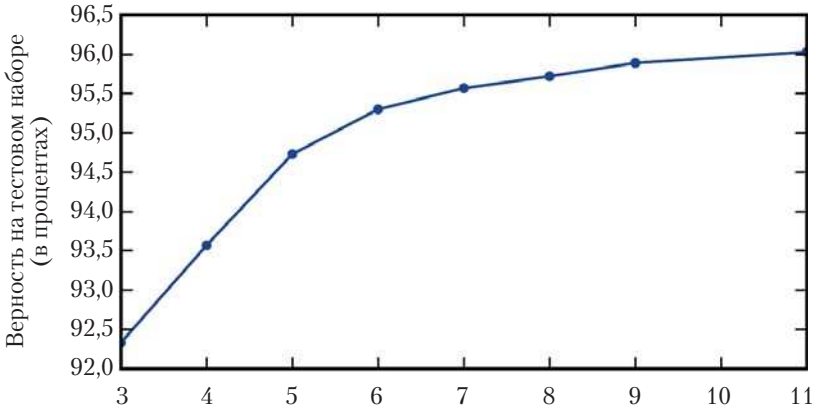

Рис. 6.6 ❖ Влияние глубины. Эмпирические результаты показывают, что более глубокие сети лучше обобщаются в задаче распознавания нескольких цифр на фотографиях адресов. Данные взяты из работы Goodfellow et al. (2014d). Верность на тестовом наборе монотонно растет при увеличении глубины. На рис. 6.7 приведены результаты контрольного эксперимента, показывающие, что увеличение других размерных характеристик модели не дает такого же эффекта

6.4.2. Другие архитектурные подходы

До сих пор мы описывали нейронную сеть как состоящую из простой цепочки слоев, а основными параметрами были глубина сети и ширина каждого слоя. На практике нейронные сети куда более разнообразны.

Многие архитектуры нейронных сетей разрабатывались под конкретные задачи. В главе 9 описаны сверточные сети – специальные архитектуры, применяемые в задачах компьютерного зрения. Сети прямого распространения также обобщаются на рекуррентные нейронные сети для обработки последовательностей (глава 10), у которых имеются собственные архитектурные особенности.

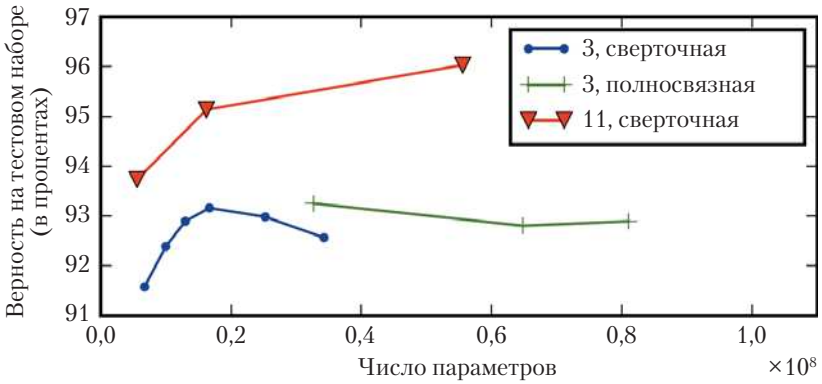

Рис. 6.7 ❖ Влияние числа параметров. Более глубокие модели обычно работают лучше. Но это не просто потому, что модель больше. Эксперимент, описанный в работе Goodfellow et al. (2014d), показывает, что увеличение числа параметров в слоях сверточной сети, не сопровождаемое увеличением ее глубины, далеко не так эффективно с точки зрения обобщения на тестовый набор. В надписях на рисунке указана глубина сети, соответствующей каждой кривой, и что именно отражает кривая: изменение размера сверточных или полносвязных слоев. Мы видим, что мелкие модели в этом контексте оказываются переобучены, когда число параметров составляет примерно 20 миллионов, а качество глубоких продолжает улучшаться вплоть до числа параметров порядка 60 миллионов. Это позволяет предположить, что глубокая модель выражает полезную гипотезу о пространстве обучаемых ей функций, а именно она выражает веру в то, функция должна быть образована композицией многих более простых функций. Результатом может быть либо обучение представления, составленного из более простых представлений (например, углы, определяемые в терминах границ), либо обучение программы, состоящей из последовательных зависимых друг от друга шагов (например, сначала найти множество объектов, затем сегментировать их, отделив друг от друга, и потом распознать их)

В общем случае слои необязательно должны быть соединены в цепочку, хотя это наиболее распространенная практика. Во многих архитектурах строится главная цепочка, а затем на нее накладываются специальные свойства, например прямые связи (skip connections), ведущие от слоя i к слою $i + 2$ и выше. Благодаря таким связям упрощается распространение градиента от выходных слоев к слоям, расположенным ближе к входу.

Еще один ключевой архитектурный вопрос – как именно соединяются между собой пары слоев. В стандартном слое нейронной сети, описываемом матрицей линейного преобразования $\boldsymbol{W}$, каждый входной блок соединен с каждым выходным. Но во многих специальных сетях, описываемых в следующих главах, число соединений меньше, т. е. каждый блок входного слоя соединен лишь с небольшим подмножеством блоков выходного слоя. Такие стратегии позволяют уменьшить число параметров и объем вычислений, необходимых для обсчета сети, но зачастую сильно зависят от характера задачи. Например, в сверточных сетях (глава 9) применяется особая структура разреженных соединений, весьма эффективная в задачах компьютерного зрения. В этой главе трудно дать более конкретный совет касательно архитектуры нейронной сети общего вида. В последующих главах мы разработаем архитектурные стратегии для частных случаев, доказавшие свою эффективность в различных предметных областях.

6.5. Обратное распространение и другие алгоритмы дифференцирования

При использовании нейронной сети прямого распространения, которая принимает вход $\boldsymbol{x}$ и порождает выход $\hat{\boldsymbol{y}}$, информация передается по сети в одном направлении – вперед. Вход $\boldsymbol{x}$ содержит начальную информацию, которая доходит до скрытых блоков каждого слоя, и в конечном итоге порождается $\hat{\boldsymbol{y}}$. Это и называется **прямым распространением**. На этапе обучения прямое распространение может продолжаться, пока не будет получена скалярная стоимость $J(\boldsymbol{\theta})$. Алгоритм **обратного распространения** (Rumelhart et al., 1986a) позволяет передавать информацию о стоимости обратно по сети для вычисления градиента.

Аналитически выписать выражение для градиента легко, но его численное вычисление может оказаться накладным. В алгоритме обратного распространения для этого применяется простая и недорогая процедура.

Сам термин «обратное распространение» зачастую трактуют неверно, понимая под ним весь алгоритм обучения многослойных нейронных сетей. На самом деле обратное распространение относится только к вычислению градиента, тогда как для обучения с помощью этого градиента применяют другие алгоритмы, например стохастического градиентного спуска. Кроме того, иногда ошибочно полагают, что обратное распространение касается только многослойных нейронных сетей, хотя, в принципе, так можно вычислять производные любой функции (для некоторых функций правильным ответом является сообщение о том, что производная не определена). Мы опишем, как вычислить градиент $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ произвольной функции f, где $\boldsymbol{x}$ – множество аргументов, по которым производные вычисляются, а $\boldsymbol{y}$ – дополнительное множество аргументов, по которым они не вычисляются. В алгоритмах обучения нас чаще всего интересует градиент функции стоимости относительно ее параметров $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Во многих задачах машинного обучения приходится вычислять и другие производные – в процессе обучения или для анализа обученной модели. Алгоритм обратного распространения можно применить и к таким задачам тоже. Идея вычисления производных путем распространения информации по сети очень общая, она применима, в частности, к вычислению якобиана функции f, порождающей много значений. Но мы ограничимся наиболее типичным случаем, когда значением f является скаляр.

6.5.1. Графы вычислений

До сих пор мы использовали при обсуждении нейронных сетей сравнительно неформальный язык графов. Но для точного описания алгоритма обратного распространения полезно иметь более формальный язык **графов вычислений**.

Есть много способов формализации вычислений в виде графов.

В данном случае вершины графа будут соответствовать переменным. Переменная может быть скаляром, вектором, матрицей, тензором или иметь еще какой-то тип.

Для формализации графов нам понадобится также ввести понятие **операции**. Операция – это простая функция одной или многих переменных. Язык графов сопровождается множеством допустимых операций. Функции, более сложные, чем операции, можно описать с помощью композиции операций.

Без ограничения общности можно считать, что операция возвращает единственную переменную. Общность не теряется, потому что эта переменная может состоять

из нескольких элементов, как, например, вектор. Программные реализации алгоритма обратного распространения обычно поддерживают операции с несколькими выходами, но в нашем описании мы отказываемся от этого, чтобы не погрязнуть в непринципиальных для понимания деталях.

Если переменная y вычисляется применением некоторой операции к переменной x, то мы проводим в графе ориентированное ребро от x к y. Иногда выходная вершина аннотируется именем примененной операции, а иногда эта метка опускается, если операция понятна из контекста.

На рис. 6.8 приведены примеры графов вычислений.

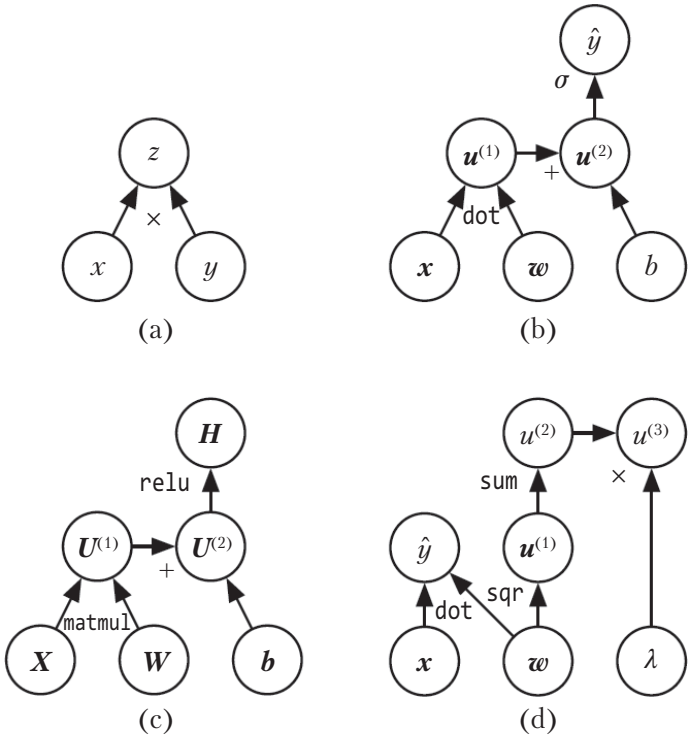

Рис. 6.8 ❖ Примеры графов вычислений. (*a*) Граф, в котором операция $\times$ используется для вычисления $z = xy$. (*b*) Граф вычисления логистической регрессии $\hat{y} = \sigma(\boldsymbol{x}^\top \boldsymbol{w} + b)$. Некоторые промежуточные выражения не поименованы в алгебраическом выражении, но должны иметь имена в графе. Мы просто обозначаем i-ю переменную такого рода $\boldsymbol{u}^{(i)}$. (*c*) Граф вычисления выражения $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$, который вычисляет матрицу плана $\boldsymbol{H}$, содержащую блоки линейной ректификации, если дана матрица плана, содержащая мини-пакет входов $\boldsymbol{X}$. (*d*) В примерах a–c к каждой переменной применялось не более одной операции, но это необязательно. Здесь показан граф вычислений, в котором применяется более одной операции к весам $\boldsymbol{w}$ модели линейной регрессии. Веса используются как для вычисления предсказания $\hat{y}$, так и для вычисления штрафа за сложность $\lambda \Sigma_i w_i^2$, поощряющего снижение весов

6.5.2. Правило дифференцирования сложной функции

Это правило применяется для вычисления производных функций, являющихся композициями других функций, чьи производные известны. Алгоритм обратного распространения как раз и посвящен реализации таких вычислений путем очень эффективного упорядочения операций.

Пусть x – вещественное число, а f и g – функции, отображающие одно вещественное число в другое. Обозначим $y = g(x)$, $z = f(g(x)) = f(y)$. Тогда правило дифференцирования сложной функции записывается в виде:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

Мы можем обобщить эту формулу на случай нескольких переменных. Пусть $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g отображает $\mathbb{R}^m$ в $\mathbb{R}^n$, а f отображает $\mathbb{R}^n$ в $\mathbb{R}$. Если $\boldsymbol{y} = g(\boldsymbol{x})$, $z = f(\boldsymbol{y})$, то

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

В векторной нотации эту формулу можно записать так:

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\mathsf{T}} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

где $\partial \boldsymbol{y}/\partial \boldsymbol{x}$ – матрица Якоби функции g размера $n \times m$.

Отсюда видно, что градиент по переменной $\boldsymbol{x}$ можно получить, умножив матрицу Якоби $\partial \boldsymbol{y}/\partial \boldsymbol{x}$ на градиент $\nabla_{\boldsymbol{y}} z$. Алгоритм обратного распространения заключается в вычислении такого произведения якобиана на градиент для каждой операции в графе.

Обычно алгоритм обратного распространения применяется к тензорам произвольной размерности, а не просто к векторам. Концептуально это то же самое, что применение к векторам. Разница только в том, как числа организуются в сетке для представления тензора. Можно вообразить, что тензор сериализуется в вектор перед обратным распространением, затем вычисляется векторнозначный градиент, после чего градиент снова преобразуется в тензор. В таком переупорядоченном представлении обратное распространение – это все то же умножение якобиана на градиенты.

Для обозначения градиента значения z относительно тензора $\mathbf{X}$ мы пишем $\nabla_{\mathbf{X}} z$, как если бы $\mathbf{X}$ был просто вектором. Теперь индексы элементов $\mathbf{X}$ составные – например, трехмерный тензор индексируется тремя координатами. Мы можем абстрагироваться от этого различия, считая, что одна переменная i представляет целый кортеж индексов. Для любого возможного индексного кортежа i $(\nabla_{\mathbf{X}} z)_i$ обозначает частную производную $\partial z/\partial \mathbf{X}_i$ – точно так же, как для любого целого индекса i $(\nabla_{\boldsymbol{x}} z)_i$ обозначает $\partial z/\partial x_i$. В этих обозначениях можно записать правило дифференцирования сложной функции в применении к тензорам. Если $\mathbf{Y} = g(\mathbf{X})$ и $z = f(\mathbf{Y})$, то

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j)\frac{\partial z}{\partial Y_j}. \tag{6.47}$$

6.5.3. Рекурсивное применение правила дифференцирования сложной функции для получения алгоритма обратного распространения

С помощью правила дифференцирования сложной функции легко записать алгебраическое выражение для градиента скаляра относительно любой вершины графа вычислений, приведшей к этому скаляру. Но при вычислении этого выражения компьютером возникают дополнительные вопросы.

Точнее, в выражении градиента могут многократно встречаться некоторые подвыражения. Процедура вычисления градиента должна решить, следует ли эти подвыражения хранить или каждый раз вычислять заново. На рис. 6.9 показано, как такие подвыражения могут возникать. В некоторых случаях вычисление одного и того же подвыражения дважды – расточительство. В сложных графах количество таких бессмысленных вычислений может расти экспоненциально, в результате чего наивная реализация правила дифференцирования сложной функции становится невозможной. А иногда повторное вычисление одного подвыражения является способом уменьшить потребление памяти за счет увеличения времени работы.

Начнем с варианта алгоритма обратного распространения, в котором порядок вычисления градиента задается непосредственно (алгоритм 6.2 в сочетании с алгоритмом 6.1 ассоциированного прямого вычисления) – путем рекурсивного применения правила дифференцирования сложной функции. Эти вычисления можно выполнить напрямую или считать описание алгоритма символической спецификацией графа вычислений обратного распространения. Однако в этой формулировке отсутствует явное построение символического графа манипуляций, необходимых для вычисления градиента. Такая формулировка приведена в разделе 6.5.6 – это алгоритм 6.5, заодно обобщенный на случай вершин, содержащих произвольные тензоры.

Сначала рассмотрим граф вычислений, описывающий, как вычислить один скаляр $u^{(n)}$ (скажем, потерю на обучающем примере). Это та величина, для которой мы хотим получить градиент относительно n_i входных вершин от $u^{(1)}$ до $u^{(n_i)}$. Иными словами, мы хотим вычислить $\partial u^{(n)}/\partial u^{(i)}$ для всех $i \in \{1, 2, ..., n_i\}$. Если алгоритм обратного распространения применяется к вычислению градиентов для градиентного спуска по параметрам, то $u^{(n)}$ – стоимость, ассоциированная с примером или мини-пакетом, а $u^{(1)}$, ..., $u^{(n_i)}$ соответствуют параметрам модели.

Будем предполагать, что вершины графа упорядочены таким образом, что можно вычислять их выходы один за другим, начиная с $u^{(n_i+1)}$ и до $u^{(n)}$. В алгоритме 6.1 предполагается, что с вершиной $u^{(i)}$ ассоциирована операция $f^{(i)}$, а вычисления в ней производятся по формуле

$$u^{(i)} = f(\mathbb{A}^{(i)}), \tag{6.48}$$

где $\mathbb{A}^{(i)}$ – множество всех вершин, являющихся родителями $u^{(i)}$.

Этот алгоритм определяет вычисления при прямом распространении, которые можно было бы поместить в граф $\mathcal{G}$. Для выполнения обратного распространения мы можем построить граф вычислений, который зависит от $\mathcal{G}$ и добавляет дополнительные вершины. Эти вершины образуют подграф $\mathcal{B}$, содержащий по одной вершине на каждую вершину $\mathcal{G}$. Вычисления в $\mathcal{B}$ производятся в порядке, обратном вычислениям

в $\mathcal{G}$, а каждая вершина $\mathcal{B}$ вычисляет производную $\partial u^{(n)}/\partial u^{(i)}$, ассоциированную с вершиной $u^{(i)}$ графа прямого распространения. Делается это с помощью правила дифференцирования сложной функции применительно к скалярному выходу $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j\in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}, \tag{6.49}$$

как описано в алгоритме 6.2. Подграф $\mathcal{B}$ содержит ровно одно ребро для каждого ребра из вершины $u^{(j)}$ в вершину $u^{(i)}$ графа $\mathcal{G}$. Ребро из $u^{(j)}$ в $u^{(i)}$ ассоциировано с вычислением $\partial u^{(i)}/\partial u^{(j)}$. Кроме того, для каждой вершины вычисляется скалярное произведение между уже вычисленным градиентом относительно вершин $u^{(i)}$, являющихся непосредственными потомками $u^{(j)}$, и вектором, содержащим частные производные $\partial u^{(i)}/\partial u^{(j)}$ для тех же дочерних вершин $u^{(i)}$. Таким образом, объем вычислений в алгоритме обратного распространения линейно зависит от числа ребер графа $\mathcal{G}$, а обработка каждого ребра состоит в вычислении частной производной (одной вершины по одной из ее родительских вершин), одного умножения и одного сложения. Ниже мы обобщим этот анализ на тензорнозначные вершины, цель которых – сгруппировать несколько скалярных значений в одной вершине и повысить эффективность реализации.

Алгоритм 6.1. Процедура, которая выполняет вычисления, отображающие n_i входов $u^{(1)}$, ..., $u^{(n_i)}$ в выход $u^{(n)}$. Она определяет граф вычислений, в котором каждая вершина вычисляет числовое значение $u^{(i)}$ путем применения функции $f^{(i)}$ ко множеству аргументов $\mathbb{A}^{(i)}$, содержащему значения предыдущих вершин $u^{(j)}$, $j < i$ и $j \in Pa(u^{(i)})$. Входом для графа вычислений является вектор $\boldsymbol{x}$, хранящийся в первых n_i вершинах $u^{(1)}$, ..., $u^{(n_i)}$. Результат графа вычислений читается из последней (выходной) вершины $u^{(n)}$.

for $i = 1, ..., n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, ..., n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

Алгоритм обратного распространения призван уменьшить количество общих подвыражений без учета доступной памяти. Точнее, он вычисляет порядка одного произведения якобиана на вектор на каждую вершину графа. Это видно из того, что алгоритм 6.2 посещает каждое ребро из вершины $u^{(j)}$ в вершину $u^{(i)}$ графа ровно один раз для вычисления ассоциированной с ним частной производной $\partial u^{(i)}/\partial u^{(j)}$. Таким образом, алгоритм обратного распространения предотвращает экспоненциальный рост повторяющихся подвыражений. Другие алгоритмы могут еще уменьшить число подвыражений путем упрощения графа вычислений, а могут сэкономить память, повторно вычисляя некоторые подвыражения вместо их сохранения. Мы вернемся к этим идеям, после того как опишем сам алгоритм обратного распространения.

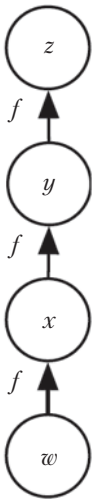

Рис. 6.9 ❖ Граф вычислений градиента, в котором возникают повторяющиеся подвыражения. Обозначим $w \in \mathbb{R}$ – вход графа. В качестве операции, применяемой на каждом шаге цепочки, используем ту же функцию $f : \mathbb{R} \longrightarrow \mathbb{R}$, т. е. $x = f(w)$, $y = f(x)$, $z = f(y)$. Для вычисления $\partial z/\partial w$ применяем уравнение 6.44 и получаем:

$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w). \tag{6.53}$$

Уравнение (6.52) подсказывает реализацию: вычислить значение $f(w)$ один раз и сохранить его в переменной x. Именно такой подход применяется в алгоритме обратного распространения. Альтернативный подход подсказан уравнением (6.53), в котором подвыражение $f(w)$ встречается более одного раза. В этом случае $f(w)$ заново вычисляется всякий раз, как понадобится. Если памяти для хранения значений подвыражений достаточно, то подход, подсказанный уравнением (6.52), очевидно, предпочтительнее, т. к. требует меньше времени. Но и уравнение (6.53) – допустимая реализация правила дифференцирования сложной функции, полезная, когда память ограничена.

Алгоритм 6.2. Упрощенный вариант алгоритма обратного распространения для вычисления производных $u^{(n)}$ по переменным в графе. Этот пример позволит лучше понять алгоритм в простом случае, когда все переменные – скаляры, а мы хотим вычислить производные по $u^{(1)}$, ..., $u^{(n_i)}$. Вычислительная сложность этого алгоритма пропорциональна числу ребер графа в предположении, что вычисление частной производной, ассоциированной с каждым ребром, занимает постоянное время. Порядок тот же, что для объема вычислений в алгоритме прямого распространения. Каждая производная $\partial u^{(i)}/\partial u^{(j)}$ является функцией от родителей $u^{(j)}$ вершины $u^{(i)}$, и таким образом вершины прямого графа связываются с добавленными в граф обратного распространения.

Выполнить алгоритм прямого распространения (алгоритм 6.1) для вычисления функций активации сети.
Инициализировать `grad_table`, структуру данных, в которой будут храниться вычисленные производные. В элементе `grad_table`$[u^{(i)}]$ хранится вычисленное значение $\partial u^{(n)}/\partial u^{(i)}$.
`grad_table`$[u^{(n)}] \leftarrow 1$
for $j = n - 1$ down to 1 **do**
В следующей строке вычисляется $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j\in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ с использованием сохраненных значений: `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j\in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$.
end for
return {`grad_table`$[u^{(i)}] \mid i = 1, ..., n_i$}

6.5.4. Вычисление обратного распространения в полносвязном МСП

Чтобы прояснить данное выше определение вычисления обратного распространения, рассмотрим конкретный граф, ассоциированный с полносвязным многослойным перцептроном.

Сначала в алгоритме 6.3 показано прямое распространение, которое отображает параметры на потерю обучения с учителем $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$, ассоциированную с одним обучающим примером $(\boldsymbol{x}, \boldsymbol{y})$, где $\hat{\boldsymbol{y}}$ – выход нейронной сети, на вход которой подан вектор $\boldsymbol{x}$.

Затем в алгоритме 6.4 показано соответствующее вычисление, в котором к этому графу применяется алгоритм обратного распространения.

Алгоритмы 6.3 и 6.4 – демонстрации для лучшего понимания идеи. Они специализированы для решения одной конкретной задачи.

Современные программные реализации основаны на обобщенной форме обратного распространения, описанной в разделе 6.5.6 ниже, они применимы к любому графу вычислений, поскольку явно манипулируют структурой данных для представления символьных вычислений.

Алгоритм 6.3. Прямое распространение по типичной глубокой нейронной сети и вычисление функции стоимости. Потеря $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ зависит от выхода $\hat{\boldsymbol{y}}$ и метки $\boldsymbol{y}$ (см. раздел 6.2.1.1, где приведены примеры функций потерь). Для получения полной стоимости J потерю можно сложить с регуляризатором $\Omega(\theta)$, где θ содержит все параметры (веса и смещения). Алгоритм 6.4 показывает, как вычислить градиенты J относительно параметров $\boldsymbol{W}$ и $\boldsymbol{b}$. Для простоты в этой демонстрации есть только один входной пример $\boldsymbol{x}$. На практике следует использовать мини-пакет. Более реалистичная демонстрация приведена в разделе 6.5.7.

Require: глубина сети l
Require: $\boldsymbol{W}^{(i)}$, $i \in \{1, ..., l\}$, матрица весов модели
Require: $\boldsymbol{b}^{(i)}$, $i \in \{1, ..., l\}$, смещения модели
Require: $\boldsymbol{x}$, входа процесса
Require: $\boldsymbol{y}$, метки
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, ..., l$ **do**

$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$

Алгоритм 6.4. Вычисление обратного распространения для глубокой нейронной сети из алгоритма 6.3, в котором помимо входа $\boldsymbol{x}$ используются метки $\boldsymbol{y}$. В результате вычисления получаются градиенты функций активации $\boldsymbol{a}^{(k)}$ для каждого слоя k, начиная с выходного и до первого скрытого слоя. Эти градиенты можно интерпретировать как указания о том, как следует изменить выход каждого слоя, чтобы уменьшить ошибку, и, зная их, мы можем вычислить градиенты параметров каждого слоя. Градиенты весов и смещений можно тут же использовать для обновления стохастического градиента (обновление выполняется сразу после вычисления градиентов) или в составе других градиентных методов оптимизации.

После вычисления прямого распространения вычислить градиент выходного слоя:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ **do**
Преобразовать градиент выходного слоя в градиент функций активации до применения нелинейности (поэлементное умножение, если f поэлементная):
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
Вычислить градиенты весов и смещений (включая член регуляризации там, где необходимо):
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda\nabla_{\boldsymbol{b}^{(k)}}\Omega(\theta)$
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\,\boldsymbol{h}^{(k-1)\top} + \lambda\nabla_{\boldsymbol{W}^{(k)}}\Omega(\theta)$
Распространить градиенты на функции активации предыдущего скрытого уровня:
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\boldsymbol{g}$
end for

6.5.5. Символьно-символьные производные

Алгебраические выражения и графы вычислений оперируют символами, т. е. переменными, не имеющими конкретного значения. Соответствующие представления называются **символьными представлениями**. В ходе использования или обучения нейронной сети мы должны приписать символам конкретные значения. Символьный вход сети $\boldsymbol{x}$ заменяется **числовым** значением, например $[1.2, 3.765, -1.8]^\top$.

В некоторых подходах к обратному распространению берутся граф вычислений и множество числовых значений, подаваемых на вход графа, а возвращается множество числовых значений, равных градиентам во входных точках. Такой подход мы называем **символьно-числовым дифференцированием**. Он реализован в библиотеках Torch (Collobert et al., 2011b) и Caffe (Jia, 2013).

Другой подход – взять граф вычислений и добавить в него дополнительные вершины, содержащие символьное описание требуемых производных. Он используется в библиотеках Theano (Bergstra et al., 2010; Bastien et al., 2012) и TensorFlow

(Abadi et al., 2015). Пример его работы показан на рис. 6.10. Основное преимущество такого подхода в том, что производные описываются на том же языке, что и исходное выражение. Поскольку производные – просто еще один граф вычислений, становится возможным прогнать алгоритм обратного распространения еще раз, т. е. продифференцировать производные для получения производных высшего порядка (вычисление производных высшего порядка рассматривается в разделе 6.5.10).

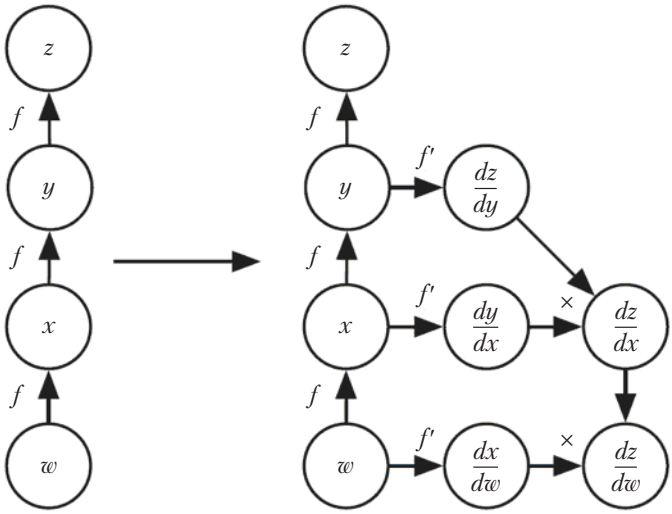

Рис. 6.10 ❖ Пример символьно-символьного подхода к вычислению производных. В этом случае алгоритму обратного распространения вообще не нужен доступ к фактическим числовым значениям. Вместо этого в граф вычислений добавляются вершины, описывающие, как вычислять производные. Впоследствии универсальный движок обсчета графа может вычислить производные для любых конкретных значений. (*Слева*) В этом примере мы начали с графа, представляющего вычисление $z = f(f(f(w)))$. (*Справа*) Мы выполнили алгоритм обратного распространения, потребовав построить граф для выражения, соответствующего dz/dw. Здесь мы не объясняем, как работает алгоритм обратного распространения. Единственная цель – показать желаемый результат: это граф вычислений, содержащий символьное описание производной

Мы будем пользоваться вторым подходом и опишем алгоритм обратного распространения в терминах построения графа вычислений производных. Впоследствии любое подмножество графа можно вычислить, подставив конкретные числовые значения. Это позволяет не указывать точно, в какой момент должна быть вычислена каждая операция, поскольку универсальный движок обсчета графа умеет вычислять каждую вершину, как только становятся доступными значения ее родителей.

Описанный символьно-символьный подход включает в себя символьно-числовой. Последний можно рассматривать как выполнение в точности тех же вычислений, ка-

кие выполняет граф, построенный в первом случае. Основное различие состоит в том, что при символьно-числовом подходе этот граф явно не создается.

6.5.6. Общий алгоритм обратного распространения

Алгоритм обратного распространения очень прост. Чтобы вычислить градиент некоторой скалярной величины z относительно одного из предков в графе, мы прежде всего заметим, что градиент по z равен $dz/dz = 1$. Затем можно вычислить градиент по каждому родителю z в графе, умножая текущий градиент на якобиан операции, породившей z. И так мы продолжаем умножать на якобианы, двигаясь в обратном направлении по графу, пока не дойдем до $\boldsymbol{x}$. Если некоторая вершина достижима из z по двум или более путям, то мы просто суммируем градиенты, полученные вдоль каждого пути.

Более формально, каждая вершина графа $\mathcal{G}$ соответствует некоторой переменной. Для максимальной общности будем считать, что эта переменная описывается тензором $\mathbf{V}$. В общем случае число размерностей тензора произвольно. Скаляры, векторы и матрицы являются частными случаями тензоров.

Будем предполагать, что с каждой переменной $\mathbf{V}$ ассоциированы следующие подпрограммы:

- `get_operation`($\mathbf{V}$): возвращает операцию, которая вычисляет $\mathbf{V}$, представляя ее в виде ребер графа вычислений, входящих в $\mathbf{V}$. Например, на языке C++ (или Python) можно написать класс, представляющий операцию умножения матриц, и функцию `get_operation`. Предположим, что некоторая переменная является результатом умножения матриц, $\boldsymbol{C} = \boldsymbol{AB}$. Тогда `get_operation`($\mathbf{V}$) возвращает указатель на экземпляр соответствующего класса C++.
- `get_consumers`($\mathbf{V}$, $\mathcal{G}$): возвращает список переменных, представленных дочерними вершинами $\mathbf{V}$ в графе вычислений $\mathcal{G}$.
- `get_inputs`($\mathbf{V}$, $\mathcal{G}$): возвращает список переменных, представленных родительскими вершинами $\mathbf{V}$ в графе вычислений $\mathcal{G}$.

С каждой операцией `op` ассоциированна также операция `bprop`. Она умеет вычислять произведение якобиана на вектор, описываемое формулой (6.47). Именно так алгоритм обратного распространения достигает максимальной общности. Каждая операция знает, как выполнить обратное распространение по ребрам графа, в которых она участвует. Например, для создания переменной $\boldsymbol{C} = \boldsymbol{AB}$ используется операция умножения матриц. Пусть градиент скалярной величины z относительно $\boldsymbol{C}$ равен матрице $\boldsymbol{G}$. Тогда операция матричного умножения отвечает за определение двух правил обратного распространения, по одному для каждого аргумента. Если мы знаем, что градиент по выходу равен $\boldsymbol{G}$, то метод `bprop` операции умножения матриц должен сказать, что градиент относительно $\boldsymbol{A}$ равен $\boldsymbol{GB}^\top$. Аналогично, если запросить у метода `bprop` градиент относительно $\boldsymbol{B}$, то операция умножения матриц, ответственная за реализацию `bprop`, сообщит, что нужный нам градиент равен $\boldsymbol{A}^\top\boldsymbol{G}$. Сам метод обратного распространения ничего не знает о правилах дифференцирования. Он только вызывает методы `bprop` каждой операции, передавая им нужные аргументы. Формально метод `op.bprop(inputs,`$\mathbf{X}$`,`$\mathbf{G}$`)` должен вернуть

$$\sum_i (\nabla_{\mathbf{X}} \texttt{op.f(inputs)}_i)\boldsymbol{G}_i, \tag{6.54}$$

а это и есть реализация правила дифференцирования сложной функции, выраженной уравнением (6.47). Здесь `inputs` – это список входов операции, `op.f` – математическая

функция, реализуемая операцией, **X** – входы, градиенты которых мы хотим вычислить, а **G** – градиент по выходу операции.

Метод `op.bprop` всегда должен считать, что все его входы отличны друг от друга, даже если на самом деле это не так. Например, если оператору `mul` передаются две копии x для вычисления x^2, то метод `op.bprop` все равно должен вернуть x в качестве производной по обоим входам. Впоследствии алгоритм обратного распространения сложит оба аргумента и получит $2x$ – правильную полную производную по x.

Программные реализации алгоритма обратного распространения обычно предоставляют как операции, так и их методы `bprop`, поэтому пользователь библиотеки глубокого обучения может выполнить обратное распространение по графам, построенным из таких общеупотребительных операций, как умножение матриц, потенцирование, логарифмирование и т. д. Программисты, создающие новые реализации обратного распространения, а также опытные пользователи, которым нужно добавить свои операции в дополнение к существующим в библиотеке, должны реализовать методы `op.bprop` новых операций самостоятельно.

Формально алгоритм обратного распространения описан в алгоритме 6.5.

Алгоритм 6.5. Внешняя обвязка алгоритма обратного распространения. Здесь производятся только инициализация и очистка. Основная работа выполняется в подпрограмме `build_grad`, описанной в алгоритме 6.6.

Require: $\mathbb{T}$, множество переменных, градиенты которых нужно вычислить.
Require: $\mathcal{G}$, граф вычислений.
Require: z, переменная, подлежащая дифференцированию
  Обозначим $\mathcal{G}'$ часть графа $\mathcal{G}$, содержащую только вершины, являющиеся предками z и потомками вершин из $\mathbb{T}$.
  Инициализировать `grad_table`, структуру данных, ассоциирующую тензоры с их градиентами
  `grad_table`[z] $\leftarrow 1$
  for **V** in $\mathbb{T}$ **do**
    `build_grad`(**V**, $\mathcal{G}$, $\mathcal{G}'$, `grad_table`)
  end for
  Return ограничение `grad_table` на $\mathbb{T}$

Алгоритм 6.6. Подпрограмма `build_grad`(**V**, $\mathcal{G}$, $\mathcal{G}'$, `grad_table`), вызываемая в цикле алгоритма обратного распространения 6.5

Require: **V**, переменная, градиент которой следует добавить в $\mathcal{G}$ и `grad_table`
Require: $\mathcal{G}$, модифицируемый граф
Require: $\mathcal{G}'$, ограничение $\mathcal{G}$ на вершины, участвующие в вычислении градиента
Require: `grad_table`, структура данных, отображающая вершины на их градиенты
  if **V** находится в `grad_table` **then**
    Return `grad_table`[**V**]
  end if
  $i \leftarrow 1$
  for **C** in `get_consumers`(**V**, $\mathcal{G}'$) **do**
    `op` $\leftarrow$ `get_operation`(**C**)
    D $\leftarrow$ `build_grad`(**C**, $\mathcal{G}$, $\mathcal{G}'$, `grad_table`)
    $\mathbf{G}^{(i)} \leftarrow$ `op.bprop`(`get_inputs`(**C**, $\mathcal{G}'$), **V**, **D**)

$i \leftarrow i + 1$
end for
$\mathbf{G} \leftarrow \Sigma_i \mathbf{G}^{(i)}$
`grad_table`[$\mathbf{V}$] = $\mathbf{G}$
Вставить $\mathbf{G}$ и участвовавшие в его создании операции в $\mathcal{G}$
Return G

В разделе 6.5.2 мы объяснили, что алгоритм обратного распространения был разработан, чтобы избежать многократного вычисления одного и того же выражения при дифференцировании сложной функции. Из-за таких повторов время выполнения наивного алгоритма могло расти экспоненциально. Теперь, описав алгоритм обратного распространения, мы можем оценить его вычислительную сложность. Если предположить, что стоимость вычисления всех операций приблизительно одинакова, то вычислительную сложность можно проанализировать в терминах количества выполненных операций. Следует помнить, что под операцией мы понимаем базовую единицу графа вычислений, в действительности она может состоять из нескольких арифметических операций (например, умножение матриц может считаться одной операцией в графе). Вычисление градиента в графе с n вершинами никогда не приводит к выполнению или сохранению результатов более $O(n^2)$. Здесь мы подсчитываем операции в графе вычислений, а не отдельные аппаратные операции, поэтому важно понимать, что время выполнения разных операций может значительно различаться. Например, умножение двух матриц, содержащих миллионы элементов, может считаться одной операцией в графе. Легко видеть, что для вычисления градиента требуется не более $O(n^2)$ операций, потому что на этапе прямого распространения в худшем случае будут обсчитаны все n вершин исходного графа (в зависимости от того, какие значения мы хотим вычислить, может потребоваться обойти весь граф). Алгоритм обратного распространения добавляет по одному произведению якобиана на вектор, выражаемому через $O(1)$ вершин, на каждое ребро исходного графа. Поскольку граф вычислений – это ориентированный ациклический граф, число ребер в нем не более $O(n^2)$. Для типичных графов, встречающихся на практике, ситуация даже лучше. В большинстве нейронных сетей функции стоимости имеют в основном цепную структуру, так что сложность обратного распространения равна $O(n)$. Это намного лучше, чем наивный подход, при котором число обрабатываемых вершин иногда растет экспоненциально. Откуда возникает экспоненциальный рост, можно понять, раскрыв и переписав правило дифференцирования сложной функции (6.49) без рекурсии:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{по путям} (u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{из } \pi_1 = j \text{ в } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \tag{6.55}$$

Поскольку количество путей из вершины j в вершину n может экспоненциально зависеть от длины пути, то число слагаемых в этой сумме, равное числу таких путей, может расти экспоненциально с увеличением глубины графа прямого распространения. Такая высокая сложность связана с многократным вычислением $\partial u^{(i)}/\partial u^{(j)}$. Чтобы избежать повторных вычислений, мы можем рассматривать обратное распространение как алгоритм заполнения таблицы, в которой хранятся промежуточные результаты $\partial u^{(n)}/\partial u^{(i)}$. Каждой вершине графа соответствует элемент таблицы, в котором хранится градиент для этой вершины. Заполняя таблицу в определенном порядке, алгоритм обратного распространения избегает повторного вычисления мно-

гих общих подвыражений. Такую стратегию заполнения таблицы иногда называют **динамическим программированием**.

6.5.7. Пример: применение обратного распространения к обучению МСП

В качестве примера подробно проанализируем применение алгоритма обратного распространения к обучению многослойного перцептрона.

Мы разработаем очень простой многослойный перцептрон всего с одним скрытым слоем. Для обучения этой модели будем использовать метод стохастического градиентного спуска с мини-пакетом. Алгоритм обратного распространения вычисляет градиент функции стоимости на одном мини-пакете. Точнее, мы используем мини-пакет примеров из обучающего набора, представленного в виде матрицы плана $\boldsymbol{X}$ и вектора ассоциированных меток класса $\boldsymbol{y}$. Сеть вычисляет слой скрытых признаков $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$. Для простоты в этой модели отсутствуют смещения. Мы предполагаем, что язык графов включает операцию `relu`, которая поэлементно вычисляет $\max\{0, \boldsymbol{Z}\}$. Затем выдаются предсказания в виде ненормированных логарифмов вероятностей классов $\boldsymbol{H}\boldsymbol{W}^{(2)}$. Мы предполагаем также, что язык графов включает операцию `cross_entropy`, вычисляющую перекрестную энтропию между метками $\boldsymbol{y}$ и распределением вероятности, определяемым этими ненормированными логарифмами вероятностей. Результат вычисления перекрестной энтропии определяет стоимость J_{MLE}. Минимизация перекрестной энтропии дает оценку максимального правдоподобия классификатора. Но, чтобы сделать пример более реалистичным, мы включили еще член регуляризации. Общая стоимость равна

$$J = J_{\text{MLE}} + \lambda\left(\sum_{i,j}(W_{i,j}^{(1)})^2 + \sum_{i,j}(W_{i,j}^{(2)})^2\right) \tag{6.56}$$

и состоит из перекрестной энтропии и члена снижения весов с коэффициентом λ. Граф вычислений показан на рис. 6.11.

Граф вычислений градиента в этом примере настолько велик, что и рисовать, и читать его утомительно. Это наглядная демонстрация одного из преимуществ алгоритма обратного распространения: он автоматически генерирует градиенты, ручной вывод которых хотя и не представляет принципиальных трудностей, но требует много времени.

Мы можем составить приблизительное представление о поведении алгоритма обратного распространения, взглянув на граф прямого распространения на рис. 6.11. Для обучения нам нужно вычислить $\nabla_{\boldsymbol{W}^{(1)}} J$ и $\nabla_{\boldsymbol{W}^{(2)}} J$. Пройти назад от J к весам можно по двум путям: через стоимость перекрестной энтропии и через стоимость снижения весов. Путь через стоимость снижения весов относительно прост; он всегда вносит вклад $2\lambda\boldsymbol{W}^{(i)}$ в градиент по $\boldsymbol{W}^{(i)}$.

Другой путь – через стоимость перекрестной энтропии – несколько сложнее. Обозначим $\boldsymbol{G}$ градиент по ненормированным логарифмам вероятностей $\boldsymbol{U}^{(2)}$, предоставляемым операцией `cross_entropy`. Алгоритм обратного распространения должен теперь исследовать две ветви. На более короткой он прибавляет $\boldsymbol{H}^\top\boldsymbol{G}$ к градиенту $\boldsymbol{W}^{(2)}$, применяя правило обратного распространения для второго аргумента операции умножения матриц. Вторая ветвь соответствует более длинной цепочке, идущей на рисунке вниз. Сначала алгоритм вычисляет $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$, применяя правило об-

ратного распространения для первого аргумента операции матричного умножения. Затем применяется правило обратного распространения операции `relu`, чтобы обнулить компоненты градиента, соответствующие тем элементам $\boldsymbol{U}^{(1)}$, которые меньше 0. Обозначим результат $\boldsymbol{G}'$. На последнем шаге правило обратного распространения применяется ко второму аргументу операции `matmul`, в результате чего к градиенту $\boldsymbol{W}^{(1)}$ прибавляется $\boldsymbol{X}^\top \boldsymbol{G}'$.

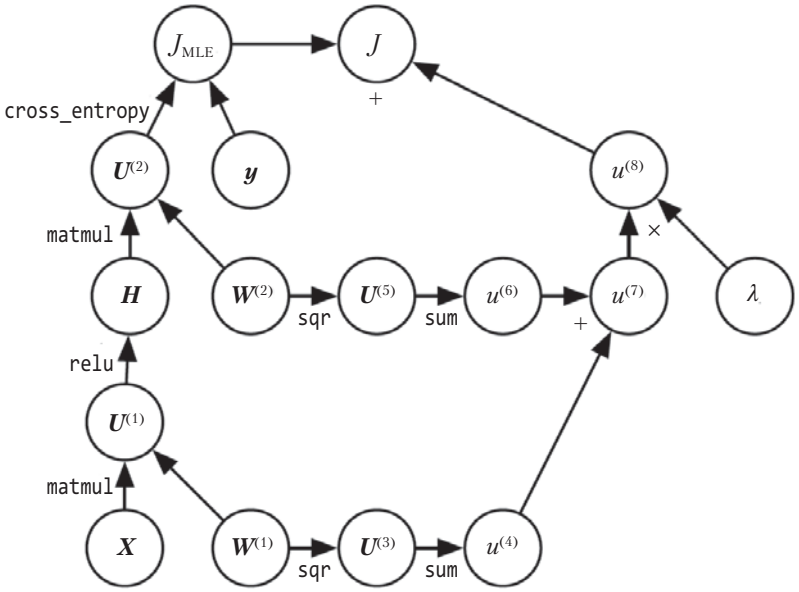

Рис. 6.11 ❖ Граф вычислений, используемый для вычисления стоимости обучения в примере МСП с одним скрытым слоем, перекрестной энтропией в качестве функции потерь и регуляризацией в форме снижения весов

После того как все градиенты вычислены, для обновления параметров можно применить алгоритм градиентного спуска или какой-нибудь другой алгоритм оптимизации.

В случае МСП вычислительная сложность определяется в первую очередь стоимостью умножения матриц. На этапе прямого распространения мы умножаем на каждую матрицу весов, что дает $O(w)$ операций сложения-умножения, где w – число весов. На этапе обратного распространения мы умножаем на транспонированные матрицы весов, стоимость этой операции такая же. Потребление оперативной памяти определяется необходимостью хранить входные аргументы для функции нелинейности скрытого слоя. Эти значения хранятся с момента вычисления и до момента, когда на обратном проходе мы вернемся в ту же точку. Таким образом, объем необходимой памяти составляет $O(mn_h)$, где m – количество примеров в мини-пакете, а n_h – количество скрытых слоев.

6.5.8. Осложнения

Наше описание алгоритма обратного распространения проще, чем в практических реализациях.

Как уже было сказано, при определении операции мы ограничились функциями, возвращающими один тензор. Большинство программных реализаций вынуждено поддерживать операции, возвращающие более одного тензора. Например, если мы хотим вычислить как максимальный элемент тензора, так и его индекс, то лучше это делать за один проход по памяти, поэтому ради эффективности процедуру следует реализовать как одну операцию с двумя выходами.

Мы не описали, как управлять потреблением памяти на этапе обратного распространения. Алгоритм обратного распространения включает сложение многих тензоров. При наивной реализации каждый тензор вычислялся бы по отдельности, а суммирование производилось бы на втором шаге. Но при этом потребляется очень много памяти, чего можно избежать, если завести один буфер и прибавлять к нему каждое значение по мере вычисления.

В практических реализациях обратного распространения необходимо также обрабатывать разные типы данных: 32- или 64-разрядные с плавающей точкой, целые и т. д. Стратегию работы с каждым типом нужно тщательно спроектировать.

В некоторых операциях встречаются неопределенные градиенты, поэтому важно отслеживать такие случаи и устанавливать, определен запрошенный пользователем градиент или нет.

Есть и другие технические детали, осложняющие дифференцирование. Они не являются непреодолимыми, в этой главе мы описали основные средства, необходимые для вычисления производных, но важно знать о существовании всяческих нюансов.

6.5.9. Дифференцирование за пределами сообщества глубокого обучения

Глубокое обучение оказалось в какой-то степени изолировано от более широкого сообщества специалистов по информатике и разрабатывало собственные традиции в отношении дифференцирования. В более общей области **автоматического дифференцирования** изучается вопрос об алгоритмическом вычислении производных. Описанный выше алгоритм обратного распространения – лишь один из подходов к автоматическому дифференцированию. Это частный случай более широкого класса методов, имеющих общее название **обратное аккумулирование** (reverse mode accumulation). В других подходах подвыражения в правиле дифференцирования сложной функции вычисляются в различном порядке. Вообще говоря, определение порядка вычислений, при котором стоимость вычислений минимальна, – трудная задача. Задача о нахождении оптимальной последовательности операций вычисления градиента является NP-полной (Naumann, 2008) в том смысле, что может потребовать упрощения алгебраических выражений до наименее затратной формы.

Например, пусть имеются переменные p_1, p_2, ..., p_n, представляющие вероятности, и переменные z_1, z_2, ..., z_n, представляющие ненормированные логарифмы вероятностей. Определим

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \tag{6.57}$$

т. е. функцию softmax, вычисление которой включает потенцирование, суммирование и деление, и построим функцию потерь на основе перекрестной энтропии $J = -\Sigma_i p_i \log q_i$. Математик заметит, что производная J по z_i имеет очень простой вид:

$q_i - p_i$. Но алгоритм обратного распространения не может упростить градиент таким образом и будет распространять градиенты через все операции логарифмирования и потенцирования, присутствующие в исходном графе. Некоторые библиотеки, например Theano (Bergstra et al., 2010; Bastien et al., 2012), умеют выполнять такого рода алгебраические подстановки, чтобы улучшить граф, предложенный чистым алгоритмом обратного распространения.

Если граф прямого распространения $\mathcal{G}$ содержит единственную выходную вершину, и каждую частную производную $\partial u^{(i)}/\partial u^{(j)}$ можно вычислить за постоянное время, то алгоритм обратного распространения гарантирует, что объем вычислений при вычислении градиентов имеет такой же порядок, как при прямом вычислении: это видно из алгоритма 6.2, поскольку в рекурсивной формулировке правила дифференцирования сложной функции (6.49) каждая локальная частная производная $\partial u^{(i)}/\partial u^{(j)}$ должна быть вычислена только один раз вместе с ассоциированным с ней умножением и сложением. Следовательно, общая вычислительная сложность составляет O(число ребер). Ее теоретически можно уменьшить, если удастся упростить граф вычислений, построенный алгоритмом обратного распространения, но это NP-полная задача. В библиотеках Theano и TensorFlow делаются попытки итеративно упростить граф, применяя эвристики, основанные на сравнении с известными типами упрощения. Мы определили обратное распространение только для вычисления градиента скалярного выхода, но это определение можно обобщить и на вычисление якобиана (либо k разных скалярных вершин графа, либо тензорнозначной вершины, содержащей k значений). Наивная реализация потребовала бы в k раз больше вычислений: для каждой внутренней скалярной вершины исходного прямого графа вычислялось бы k градиентов вместо одного. Если число выходов в графе больше числа входов, то иногда предпочтительнее другая форма автоматического дифференцирования – с **прямым аккумулированием** (forward mode accumulation). Прямое аккумулирование было предложено для вычисления градиентов в режиме реального времени в рекуррентных сетях, см., например, Williams and Zipser, 1989. При таком подходе удается также избежать хранения значений и градиентов для всего графа, пожертвовав частью вычислительной эффективности ради памяти. Связь между прямым и обратным режимами аккумулирования аналогична связи между умножением слева и справа при перемножении последовательности матриц:

$$\boldsymbol{ABCD}. \tag{6.58}$$

Можно считать, что это матрицы Якоби. Например, если $\boldsymbol{D}$ – вектор-столбец, а $\boldsymbol{A}$ содержит много строк, то в графе будет один выход и много входов. Если выполнять умножения от конца к началу, то вычислять нужно будет только произведения матрицы на вектор. Это соответствует обратному аккумулированию. Напротив, если умножать слева направо, то нужно будет вычислять произведения матрицы на матрицу, и все вычисление окажется намного дороже. Однако если число строк $\boldsymbol{A}$ меньше числа столбцов $\boldsymbol{D}$, то дешевле выполнять умножение слева направо, что соответствует прямому аккумулированию.

Во многих сообществах, не связанных с машинным обучением, принято писать код дифференцирования на традиционном языке программирования, например Python или C, и автоматически сгенерированная программа применяется к функциям,

написанным на том же языке. Но в глубоком обучении графы вычислений обычно представляются с помощью явных структур данных, создаваемых специальными библиотеками. У такого подхода есть недостаток: разработчик библиотеки должен определить методы `bprop` для каждой операции, а пользователь ограничен лишь теми операциями, которые определил автор. Однако у него есть и достоинство: для каждой операции можно написать специализированные правила обратного распространения, что позволяет повысить быстродействие и устойчивость неочевидными способами, которые автоматическая процедура вряд ли смогла бы повторить.

Таким образом, обратное распространение – не единственный и не оптимальный способ вычисления градиента, но это удобный на практике метод, который удовлетворяет потребности сообщества машинного обучения. В будущем, когда специалисты-практики будут лучше знать о достижениях общей теории автоматического дифференцирования, технология дифференцирования для глубоких сетей, возможно, усовершенствуется.

6.5.10. Производные высшего порядка

В некоторых программных системах поддерживается использование производных высшего порядка. Среди библиотек глубокого обучения этим отличаются как минимум Theano и TensorFlow. В этих библиотеках для описания производных используется та же структура данных, что для описания исходной дифференцируемой функции. Поэтому механизм символьного дифференцирования можно применить к производным.

В контексте глубокого обучения редко приходится вычислять одну вторую производную скалярной функции. Обычно нас интересуют свойства матрицы Гессе. Если имеется функция $f: \mathbb{R}^n \longrightarrow \mathbb{R}$, то ее матрица Гессе имеет размер $n \times n$. В типичном приложении машинного обучения n – это количество параметров модели, которое легко может исчисляться миллиардами. Поэтому представить матрицу Гессе целиком нереально.

Вместо явного вычисления гессиана в глубоком обучении обычно используют **методы Крылова**. Это набор итеративных приемов выполнения различных операций, например приближенного обращения матрицы или нахождения ее собственных значений и собственных векторов, с использованием только умножения матрицы на вектор.

Чтобы применить методы Крылова к гессиану, нужно только уметь вычислять произведение матрицы Гессе $\boldsymbol{H}$ и произвольного вектора $\boldsymbol{v}$. Сделать это просто (Christianson, 1992):

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}}[(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^{\top}\boldsymbol{v}]. \tag{6.59}$$

Оба градиента в этом выражении можно вычислить автоматически с помощью подходящей библиотеки. Отметим, что внешний градиент применяется к внутреннему градиенту функции.

Если сам вектор $\boldsymbol{v}$ порожден графом вычислений, то программе автоматического дифференцирования необходимо сказать, чтобы она не дифференцировала граф, порождающий $\boldsymbol{v}$.

Обычно вычислять гессиан не рекомендуется, но можно обойтись произведениями гессиана на вектор. Просто вычисляется $\boldsymbol{H}\boldsymbol{e}^{(i)}$ для всех $i = 1, ..., n$, где $\boldsymbol{e}^{(i)}$ – унитарный вектор, в котором $\boldsymbol{e}_i^{(i)} = 1$, а все остальные элементы равны 0.

6.6. Исторические замечания

Сети прямого распространения можно рассматривать как эффективный способ аппроксимации функций, основанный на использовании градиентного спуска для минимизации ошибки. С этой точки зрения, современные сети прямого распространения – кульминация многовекового развития общей теории аппроксимации.

Правило дифференцирования сложной функции, лежащее в основе алгоритма обратного распространения, было открыто в XVII веке (Leibniz, 1676; L'Hôpital, 1696). Математический анализ и алгебра с давних пор использовались для решения задач оптимизации в замкнутой форме, но метод градиентного спуска как способ итеративной аппроксимации решения задач оптимизации появился только в XIX веке (Cauchy, 1847).

Начиная с 1940-х годов, методы аппроксимации функций использовались в моделях машинного обучения, в частности в перцептроне. Однако ранние модели были линейными. Критики, в т. ч. Марвин Минский, отмечали несколько изъянов в семействе линейных моделей, например невозможность обучить функцию XOR, и это привело к отрицанию всего подхода на основе нейронных сетей.

Для обучения нелинейных функций понадобилось разработать многослойный перцептрон и средства вычисления градиентов. Эффективные применения правила дифференцирования сложной функции, основанные на динамическом программировании, начали появляться в 1960–1970-е годы, в основном в задачах автоматического управления (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Dreyfus, 1973), но также и в анализе чувствительности (Linnainmaa, 1976). В работе Werbos (1981) описано, как применить эти методы к обучению искусственных нейронных сетей. В конечном итоге идея получила практическое воплощение, после того как ее несколько раз переоткрывали в разных формах (LeCun, 1985; Parker, 1985; Rumelhart et al., 1986a). В книге «Parallel Distributed Processing» представлены результаты первых успешных экспериментов с алгоритмом обратного распространения. Ее глава, написанная Румельхартом с соавторами (Rumelhart et al., 1986b), стала значительным вкладом в популяризацию обратного распространения и ознаменовала начало периода очень активных исследований по многослойным нейронным сетям. Идеи, выдвинутые авторами книги, в особенности Румельхартом и Хинтоном, выходят далеко за рамки обратного распространения. Среди них ключевые идеи о возможности вычислительной реализации некоторых ключевых аспектов познания и обучения, получившие собирательное название «коннекционизм» из-за той важности, которую адепты этой школы придавали связям между нейронами как органами, отвечающими за обучение и память. В частности, эти идеи включают понятие распределенного представления (Hinton et al., 1986).

На волне успехов алгоритма обратного распространения исследования в области нейронных сетей обрели популярность, достигшую пика в начала 1990-х годов. Затем им на смену пришли другие методы машинного обучения, и так продолжалось до возрождения современного глубокого обучения в 2006 году.

Основные идеи, лежащие в основе сетей прямого распространения, не сильно изменились с 1980-х годов. По-прежнему используются все тот же алгоритм обратного распространения и те же подходы к градиентному спуску. Повышением качества работы, имевшим место в промежутке между 1986 и 2015 годом, нейронные сети обязаны в основном двум факторам. Во-первых, благодаря более крупным наборам

данных статистическое обобщение перестало быть серьезным препятствием на пути развития нейронных сетей. Во-вторых, сами нейронные сети стали гораздо больше благодаря более мощным компьютерам и улучшению программной инфраструктуры. Некоторые алгоритмические изменения также заметно повысили качество сетей.

Одним из таких изменений стала замена среднеквадратической ошибки семейством функций потерь на основе перекрестной энтропии. Среднеквадратическая ошибка была популярна в 1980-е и 1990-е годы, но постепенно ее вытеснили идеи перекрестной энтропии и принцип максимального правдоподобия, распространившиеся среди специалистов по статистике и машинному обучению. Использование потерь в форме перекрестной энтропии заметно повысило качество моделей с сигмоидой и softmax в роли функций активации, которые раньше – когда потери измерялись среднеквадратической ошибкой – страдали от насыщения и медленного обучения.

Еще одно важное алгоритмическое изменение, резко улучшившее качество сетей прямого распространения, – замена сигмоидных скрытых блоков кусочно-линейными, например блоками линейной ректификации. Ректификация с использованием функции $\max\{0, z\}$ появилась еще в ранних моделях нейронных сетей и восходит, по меньшей мере, к когнитрону и неокогнитрону (Fukushima, 1975, 1980). Но тогда не использовались блоки линейной ректификации, а вместо этого ректификация применялась к нелинейным функциям. Несмотря на популярность ректификации в ранних моделях, в 1980-е годы ее почти всюду заменили сигмоиды, поскольку они лучше работают в очень малых нейронных сетях. В начале 2000-х блоков линейной ректификации старательно избегали из-за какой-то суеверной боязни использовать функции активации, недифференцируемые в некоторых точках. Положение начало меняться в 2009 году. В работе Jarrett et al. (2009) было отмечено, что «использование ректифицирующей нелинейности – самый важный фактор улучшения качества системы распознавания» наряду с другими аспектами проектирования архитектуры нейронной сети.

В работе Jarrett et al. (2009) также отмечалось, что для небольших наборов данных использование ректифицирующих нелинейностей даже важнее, чем обучение весов скрытых слоев. Случайных весов достаточно для распространения полезной информации по сети с линейной ректификацией, что позволяет классифицирующему выходному слою обучаться отображению различных векторов признаков на идентификаторы классов.

Если доступно больше данных, то процесс обучения начинает извлекать так много полезных знаний, что превосходит по качеству случайным образом выбранные параметры. В работе Glorot et al. (2011a) показано, что обучение гораздо легче проходит в ректифицированных линейных сетях, чем в глубоких сетях, для которых функции активации характеризуются кривизной или двусторонним насыщением. Блоки линейной ректификации представляют также исторический интерес, поскольку демонстрируют, что нейробиология по-прежнему оказывает влияние на разработку алгоритмов глубокого обучения. В работе Glorot et al. (2011a) в обоснование блоков линейной ректификации выдвигаются биологические соображения. Ректификация на полупрямой была предложена для улавливания следующих свойств биологических нейронов: 1) для некоторых входных сигналов биологические нейроны вообще неактивны; 2) для некоторых входных сигналов выходной сигнал биологического нейрона пропорционален входному сигналу; 3) большую часть времени биологиче-

ские нейроны пребывают в состоянии неактивности (т. е. характеризуются **разреженной активацией**).

Когда в 2006 году началось возрождение глубокого обучения, сети прямого распространения по-прежнему пользовались дурной репутацией. В период с 2006 по 2012 год превалировало мнение, что такие сети не могут работать хорошо, если им не ассистируют другие модели, например вероятностные. Сегодня известно, что при наличии адекватных ресурсов и инженерных навыков сети прямого распространения работают отлично. В наши дни обучение сетей прямого распространения градиентными методами служит инструментом для разработки вероятностных моделей, как, например, вариационный автокодировщик и порождающие состязательные сети, описанные в главе 20. Начиная с 2012 года обучение сетей прямого распространения градиентными методами перестало считаться ненадежной технологией, которая должна обязательно поддерживаться другими методами. Теперь это мощная технология, применимая ко многим задачам машинного обучения. В 2006 году сообщество использовало обучение без учителя для поддержки обучения с учителем, а теперь – по иронии судьбы – все обстоит «с точностью до наоборот».

У сетей прямого распространения есть еще не раскрытый потенциал. Мы ожидаем, что в будущем они найдут применение во многих других задачах и что благодаря достижениям в разработке алгоритмов оптимизации и проектирования моделей их качество еще возрастет. В этой главе мы в общих чертах описали семейство моделей на основе нейронных сетей. А в последующих вплотную займемся их использованием – расскажем, как их регуляризировать и обучать.

Глава 7

Регуляризация в глубоком обучении

Центральная проблема машинного обучения – как создать алгоритм, который будет хорошо работать не только на обучающих, но и на новых данных. Многие используемые стратегии специально предназначены для уменьшения ошибки тестирования, быть может, за счет увеличения ошибки обучения. Эти стратегии известны под общим названием «регуляризация». В распоряжении специалиста по глубокому обучению много вариантов регуляризации. На самом деле разработка все более эффективных стратегий регуляризации – одно из основных направлений исследований в этой области.

В главе 5 были введены понятия обобщения, недообучения, переобучения, смещения, дисперсии и регуляризации. Если вы пока незнакомы с ними, ознакомьтесь с главой 5, прежде чем продолжать чтение.

В этой главе мы опишем регуляризацию подробно, уделив особое внимание стратегиям регуляризации глубоких моделей или моделей, которые используются в качестве их строительных блоков.

В некоторых разделах этой главы речь идет о стандартных концепциях машинного обучения. Если вы уже знакомы с ними, можете спокойно пропустить эти разделы. Но большая часть главы посвящена обобщению базовых концепций на случай нейронных сетей.

В разделе 5.2.2 мы определили регуляризацию как «любую модификацию алгоритма обучения, предпринятую с целью уменьшить его ошибку обобщения, не уменьшая ошибки обучения». Существует много стратегий регуляризации. В одних налагаются дополнительные ограничения на модель машинного обучения, например на значения параметров. В других в целевую функцию включаются дополнительные члены, которые можно рассматривать как мягкие ограничения на значения параметров. При правильном выборе такие дополнительные ограничения и штрафы могут приводить к повышению качества на тестовом наборе. Иногда ограничения и штрафы проектируются, чтобы выразить предпочтение более простому классу моделей и тем повысить обобщаемость. А иногда они необходимы, чтобы преобразовать недоопределенную задачу в определенную. В других вариантах регуляризации, называемых ансамблевыми методами, комбинируется несколько гипотез, объясняющих обучающие данные.

В контексте глубокого обучения большинство стратегий регуляризации основано на регуляризирующих оценках. Смысл регуляризация оценки – в увеличении сме-

щения в обмен на уменьшение дисперсии. Эффективным считается регуляризатор, который находит выгодный компромисс, т. е. значительно уменьшает дисперсию, не слишком увеличивая смещение. Обсуждая обобщение и переобучение в главе 5, мы выделили три ситуации, когда обучаемое семейство моделей либо (1) не включает истинный процесс, порождающий данные, – это соответствует недообучению и провоцирует смещение, либо (2) соответствует истинному порождающему процессу, либо (3) включает как истинный порождающий процесс, так и много других потенциальных процессов, – это режим переобучения, когда в ошибке оценивания превалирует дисперсия, а не смещение. Цель регуляризации – перевести модель из третьего режима во второй.

На практике чрезмерно сложное семейство моделей не обязательно включает не то что целевую функцию или истинный порождающий процесс, но даже хорошее приближение к тому или другому. Мы почти никогда не имеем доступа к истинному порождающему процессу, поэтому не можем знать наверняка, включает оцениваемое семейство этот процесс или нет. Однако алгоритмы глубокого обучения по большей части применяются в предметных областях, где истинный порождающий процесс почти наверняка не входит в семейство моделей. Алгоритмы глубокого обучения обычно используются в чрезвычайно сложных областях – обработка изображений, звуковых последовательностей и текстов, где истинный порождающий процесс, по существу, сводится к моделированию всего универсума. Не будет таким уж преувеличением сказать, что мы всегда пытаемся воткнуть квадратный колышек (порождающий процесс) в круглое отверстие (наше семейство моделей).

Это означает, что управление сложностью модели – не просто поиск модели правильного размера с правильным числом параметров. Вместо этого мы могли обнаружить – и в реальных приложениях глубокого обучения так почти всегда и бывает – что наилучшая эмпирическая модель (в смысле минимизации ошибки обобщения) – это большая модель, подходящим образом регуляризированная.

Далее мы дадим обзор нескольких стратегий создания большой регуляризированной глубокой модели.

7.1. Штрафы по норме параметров

Регуляризация применялась в течение десятков лет до изобретения глубокого обучения. Такие линейные модели, как линейная регрессия и логистическая регрессия, допускают простые и эффективные стратегии регуляризации.

Многие подходы к регуляризации основаны на ограничении емкости моделей (нейронных сетей, линейной регрессии или логистической регрессии) путем прибавления штрафа по норме параметра $\Omega(\boldsymbol{\theta})$ к целевой функции J. Мы будем обозначать регуляризованную целевую функцию $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}), \tag{7.1}$$

где $\alpha \in [0, \infty)$ – гиперпараметр, задающий вес члена Ω, штрафующего по норме, относительно стандартной целевой функции J. Если α равен 0, то регуляризация отсутствует. Чем больше значение α, тем сильнее регуляризация.

Минимизируя регуляризованную целевую функцию $\tilde{J}$, наш алгоритм обучения одновременно уменьшает исходную целевую функцию J на обучающих данных и некоторую меру величины параметров $\boldsymbol{\theta}$ (или какого-то подмножества параметров).

В зависимости от выбора нормы параметров предпочтительными будут те или иные решения. В этом разделе мы обсудим влияние различных норм, используемых для штрафования параметров модели.

Прежде чем с головой погрузиться в обсуждение различных норм с точки зрения регуляризации, отметим, что в нейронных сетях мы обычно предпочитаем штрафовать по норме *только веса* аффинного преобразования в каждом слое, оставляя смещения нерегуляризированными. Для точного подбора смещений, как правило, требуется меньше данных, чем для весов. Каждый вес описывает взаимодействие двух переменных. Для правильного подбора веса требуются результаты наблюдений этих переменных в разных обстоятельствах. Каждое смещение контролирует только одну переменную. Поэтому, игнорируя регуляризацию смещений, мы не слишком сильно увеличим дисперсию. Кроме того, регуляризация параметров смещения может стать причиной значительного недообучения. Далее вектор $\boldsymbol{w}$ будет обозначать все веса, на которые распространяется штраф по норме, а вектор $\boldsymbol{\theta}$ – все параметры, включая как $\boldsymbol{w}$, так и нерегуляризированные.

В контексте нейронных сетей иногда желательно использовать штрафы с разными коэффициентами α в каждом слое. Поскольку подбор правильных значений нескольких гиперпараметров иногда обходится дорого, все равно разумно использовать одно и то же снижение весов во всех слоях, просто чтобы уменьшить размер пространства поиска.

7.1.1. Регуляризация параметров по норме L^2

В разделе 5.2.2 мы уже встречали один из самых простых и распространенных видов штрафа параметров – по норме L^2, – который часто называют **снижением весов**. Цель такой стратегии регуляризации – выбирать веса, близкие к началу координат[1], за счет прибавления к целевой функции члена регуляризации $\Omega(\boldsymbol{\theta}) = {}^1\!/_2 \|\boldsymbol{w}\|_2^2$. В других научных сообществах регуляризацию по норме L^2 называют также **гребневой регрессией**, или **регуляризацией Тихонова**.

Получить качественное представление о поведении регуляризации методом снижения весов можно, изучив градиент регуляризированной целевой функции. Для простоты опустим параметр смещения, т. е. будем считать, что $\boldsymbol{\theta}$ совпадает с $\boldsymbol{w}$. Полная целевая функция в такой модели имеет вид:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = (\alpha/2)\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \tag{7.2}$$

а градиент по параметрам:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{7.3}$$

Один шаг обновления весов с целью уменьшения градиента имеет вид:

$$\boldsymbol{w} \longleftarrow \boldsymbol{w} - \varepsilon(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})). \tag{7.4}$$

---

[1] В общем случае мы могли бы путем регуляризации отдавать предпочтение параметрам вблизи любой наперед заданной точки пространства и, как ни странно, получить эффект регуляризации. Но результаты оказываются лучше, если эта точка близка к истинной, а нуль является разумным значением по умолчанию, когда мы не знаем, положительно истинное значение или отрицательно. Поскольку в большинстве случае при регуляризации стремятся сделать параметры близкими к нулю, мы будем рассматривать только этот частный случай.

То же самое можно переписать в виде:

$$\boldsymbol{w} \leftarrow (1 - \varepsilon\alpha)\boldsymbol{w} - \varepsilon\nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})). \tag{7.5}$$

Как видим, добавление члена сложения весов изменило правило обучения: теперь мы на каждом шаге умножаем вектор весов на постоянный коэффициент, меньший 1, перед тем как выполнить стандартное обновление градиента. Итак, мы описали, что происходит на одном шаге. А в целом в процессе обучения?

Еще упростим анализ, предположив квадратичную аппроксимацию целевой функции в окрестности того значения весов, при котором достигается минимальная стоимость обучения без регуляризации, $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$. Если целевая функция действительно квадратичная, как в случае модели линейной регрессии со среднеквадратической ошибкой, то такая аппроксимация идеальна. Аппроксимация $\hat{J}$ описывается формулой:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + {}^1\!/_2(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.6}$$

где $\boldsymbol{H}$ – матрица Гессе J относительно $\boldsymbol{w}$, вычисленная в точке $\boldsymbol{w}^*$. В этой квадратичной аппроксимации нет члена первого порядка, потому что $\boldsymbol{w}^*$, по определению, точка минимума, в которой градиент обращается в нуль. Из того, что $\boldsymbol{w}^*$ – точка минимума J, следует также, что матрица $\boldsymbol{H}$ положительно полуопределенная.

Минимум $\hat{J}$ достигается там, где градиент

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{7.7}$$

равен 0.

Чтобы изучить эффект снижения весов, модифицируем уравнение (7.7), прибавив градиент снижения весов. Теперь мы можем найти из него минимум регуляризированного варианта $\hat{J}$. Обозначим $\tilde{\boldsymbol{w}}$ положение точки минимума.

$$\alpha\tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0. \tag{7.8}$$

$$(\boldsymbol{H} + \alpha\boldsymbol{I})\tilde{\boldsymbol{w}} = \boldsymbol{H}\boldsymbol{w}^*. \tag{7.9}$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha\boldsymbol{I})^{-1}\boldsymbol{H}\boldsymbol{w}^*. \tag{7.10}$$

Когда α стремится к 0, регуляризированное решение $\tilde{\boldsymbol{w}}$ стремится к $\boldsymbol{w}^*$. Но что происходит, когда α возрастает? Поскольку матрица $\boldsymbol{H}$ вещественная и симметричная, мы можем разложить ее в произведение диагональной матрицы $\boldsymbol{\Lambda}$ и ортогональной матрицы собственных векторов $\boldsymbol{Q}$: $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Подставляя это разложение в уравнение (7.10), получаем

$$\tilde{\boldsymbol{w}} = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top + \alpha\boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^*, \tag{7.11}$$

$$= [\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})\boldsymbol{Q}^\top]^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^*, \tag{7.12}$$

$$= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^*. \tag{7.13}$$

Мы видим, что результатом снижения весов является масштабирование $\boldsymbol{w}^*$ вдоль направлений собственных векторов $\boldsymbol{H}$. Точнее, компонента $\boldsymbol{w}^*$, параллельная i-му собственному вектору $\boldsymbol{H}$, умножается на коэффициент $\lambda_i/(\lambda_i + \alpha)$. (Этот вид масштабирования ранее был проиллюстрирован на рис. 2.3.)

Вдоль направлений, для которых собственные значения $\boldsymbol{H}$ относительно велики, например когда $\lambda_i \gg \alpha$, эффект регуляризации сравнительно мал. Те же компоненты, для которых $\lambda_i \ll \alpha$, сжимаются почти до нуля. Это показано на рис. 7.1.

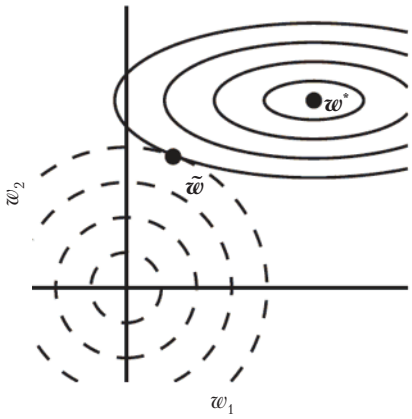

Рис. 7.1 ❖ Иллюстрация влияния регуляризации по норме L^2 (снижения весов) на значение оптимального вектора w. Сплошными эллипсами представлены линии равных значений нерегуляризированной целевой функции, а пунктирными – линии равных значений L^2-регуляризатора. В точке $\tilde{\boldsymbol{w}}$ эти конкурирующие цели достигают равновесия. По первому измерению собственное значение гессиана J мало. Целевая функция слабо растет при удалении от $\boldsymbol{w}^*$ по горизонтали. Поскольку целевая функция не выказывает сильного предпочтения этому направлению, регуляризатор дает для него значительный эффект. Регуляризатор подтягивает w_1 ближе к нулю. По второму направлению целевая функция очень чувствительна к удалению от $\boldsymbol{w}^*$. Соответствующее собственное значение велико, что указывает на сильную кривизну. Поэтому снижение весов влияет на положение w_2 сравнительно слабо

Относительно неизменными остаются только те направления, в которых параметры дают сильный вклад в уменьшение целевой функции. Если направление не дает вклада в уменьшение целевой функции, то собственное значение гессиана мало, т. е. движение в этом направлении не приводит к заметному возрастанию градиента. Компоненты вектора весов, соответствующие таким малозначимым направлениям, снижаются почти до нуля благодаря использованию регуляризации в ходе обучения.

До сих пор мы обсуждали снижение весов в терминах воздействия на оптимизацию абстрактной квадратичной функции стоимости. Но как эти эффекты проявляются конкретно в машинном обучении? Это можно выяснить на примере изучения линейной регрессии – модели, в которой истинная функция стоимости квадратичная и потому поддается проведенному выше анализу. Повторяя те же рассуждения, мы получим для этого частного случая результат, сформулированный в терминах обучающих данных. Для линейной регрессии функция стоимости равна сумме квадратов ошибок:

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}). \tag{7.14}$$

После добавления L^2-регуляризации целевая функция принимает вид:

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + {}^1\!/_2\alpha\boldsymbol{w}^\top\boldsymbol{w}. \tag{7.15}$$

В результате нормальные уравнения, из которых ищется решение:

$$\boldsymbol{w} = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y} \tag{7.16}$$

принимают вид

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}. \tag{7.17}$$

Матрица $\boldsymbol{X}^\top \boldsymbol{X}$ в уравнении (7.16) пропорциональна ковариационной матрице $(1/m)\boldsymbol{X}^\top \boldsymbol{X}$. Применение L^2-регуляризации заменяет эту матрицу на $(\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1}$ в уравнении (7.17). Новая матрица отличается от исходной только прибавлением α ко всем диагональным элементам. Диагональные элементы этой матрицы соответствуют дисперсии каждого входного признака. Таким образом, L^2-регуляризация заставляет алгоритм обучения «воспринимать» вход $\boldsymbol{X}$ как имеющий более высокую дисперсию и, следовательно, уменьшать веса тех признаков, для которых ковариация с выходными метками мала, по сравнению с добавленной дисперсией.

7.1.2. L^1-регуляризация

Регуляризация по норме L^2 – самая распространенная форма снижения весов, но есть и другие способы штрафовать за величину параметров модели. Одним из них является L^1-регуляризация.

Формально L^1-регуляризация параметров модели $\boldsymbol{w}$ определяется по формуле

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|, \tag{7.18}$$

т. е. как сумма абсолютных величин отдельных параметров[1]. Обсудим влияние L^1-регуляризации на простую модель линейной регрессии без параметра смещения – ту самую, что рассматривалась в ходе анализа L^2-регуляризации. Особенно нас интересуют различия между двумя видами регуляризации. Как и в предыдущем случае, сила регуляризации контролируется путем умножения штрафа на положительный гиперпараметр α. Таким образом, регуляризированная целевая функция $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ описывается формулой

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \|\boldsymbol{w}\|_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \tag{7.19}$$

а ее градиент (точнее, частичный градиент) равен

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \operatorname{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \tag{7.20}$$

где $\operatorname{sign}(\boldsymbol{w})$ означает, что функция sign применяется к каждому элементу $\boldsymbol{w}$.

Из уравнения (7.20) сразу видно, что эффект L^1-регуляризации совсем не такой, как L^2-регуляризации. Теперь вклад регуляризации в градиент уже не масштабируется линейно с ростом каждого w_i, а описывается постоянным слагаемым, знак которого совпадает с $\operatorname{sign}(w_i)$. Одним из следствий является тот факт, что мы уже не получим изящных алгебраических выражений квадратичной аппроксимации $J(\boldsymbol{X}; \boldsymbol{y}, \boldsymbol{w})$, как в случае L^2-регуляризации.

В нашей простой линейной модели имеется квадратичная функция стоимости, которую можно представить ее рядом Тейлора. Можно вместо этого считать, что это первые члены ряда Тейлора, аппроксимирующие функцию стоимости более сложной модели. Градиент в этой конфигурации равен

---

[1] Как и в случае L^2-регуляризации, можно было бы сдвигать параметры не к нулю, а к какому-то значению $\boldsymbol{w}^{(o)}$. Тогда L^1-регуляризация свелась бы к добавлению члена $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w} - \boldsymbol{w}^{(o)}\|_1 = \Sigma_i |w_i - w_i^{(o)}|$.

$$\nabla_{\boldsymbol{w}}\hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.21}$$

где $\boldsymbol{H}$ – снова матрица Гессе J относительно $\boldsymbol{w}$, вычисленная в точке $\boldsymbol{w}^*$.

Поскольку штраф по норме L^1 не допускает простого алгебраического выражения в случае полного гессиана общего вида, то мы сделаем еще одно упрощающее предположение: будем считать матрицу Гессе диагональной, $\boldsymbol{H} = \mathrm{diag}([H_{1,1}, ..., H_{n,n}])$, где все $H_{i,i} > 0$. Это предположение справедливо, если данные для задачи линейной регрессии были подвергнуты предварительной обработке для устранения корреляции между входными признаками, например методом главных компонент.

Нашу квадратичную аппроксимацию L^1-регуляризированной целевой функции можно представить в виде суммы по параметрам:

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = j(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} H_{i,i}(\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \alpha |w_i|\right]. \tag{7.22}$$

У задачи минимизации этой приближенной функции стоимости имеется аналитическое решение (для каждого измерения i) вида:

$$w_i = \mathrm{sign}(w_i^*)\max\left\{|w_i^*| - \frac{\alpha}{H_{i,i}}, 0\right\}. \tag{7.23}$$

Предположим, что $w_i^* > 0$ для всех i. Тогда есть два случая:

1) $w_i^* \le \alpha/H_{i,i}$. Тогда оптимальное значение w_i для регуляризированной целевой функции будет просто $w_i = 0$. Причина в том, что вклад $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ в регуляризированную целевую функцию перевешивается – в направлении i – L^1-регуляризацией, которая сдвигает значение w_i в нуль;
2) $w_i^* > \alpha/H_{i,i}$. Тогда регуляризация не сдвигает оптимальное значение w_i в нуль, а просто смещает его в этом направлении на расстояние $\alpha/H_{i,i}$.

Аналогичное рассуждение проходит, когда $w_i^* < 0$, только L^1-штраф увеличивает w_i на $\alpha/H_{i,i}$ или обращает в 0.

По сравнению с L^2-регуляризацией, L^1-регуляризация дает более **разреженное** решение. В этом контексте под разреженностью понимается тот факт, что у некоторых параметров оптимальное значение равно 0. Разреженность L^1-регуляризации является качественным отличием от поведения L^2-регуляризации. Уравнение (7.13) дает решение $\tilde{\boldsymbol{w}}$ для L^2-регуляризации. Применив к нему предположение о диагональности и положительной определенности гессиана $\boldsymbol{H}$, которое мы приняли для анализа L^1-регуляризации, найдем, что $\tilde{w}_i = \dfrac{H_{i,i}}{H_{i,i} + \alpha} w_i^*$. Если w_i^* было не равно 0, то и $\tilde{w}_i$ окажется не равным нулю. Это значит, что L^2-регуляризация не приводит к разреженности параметров, тогда как в случае L^1-регуляризации это возможно, если α достаточно велико.

Свойство разреженности, присущее L^1-регуляризации, активно эксплуатировалось как механизм **отбора признаков**, идея которого состоит в том, чтобы упростить задачу машинного обучения за счет выбора некоторого подмножества располагаемых признаков. В частности, хорошо известная модель LASSO (Tibshirani, 1995) (least absolute shrinkage and selection operator) объединяет L^1-штраф с линейной моделью и среднеквадратической функцией стоимости. Благодаря L^1-штрафу некоторые веса обращаются в 0, и соответствующие им признаки отбрасываются.

В разделе 5.6.1 мы видели, что многие стратегии регуляризации можно интерпретировать как байесовский вывод на основе оценки апостериорного максимума (MAP)

и что, в частности, L^2-регуляризация эквивалентна байесовскому выводу на основе MAP с априорным нормальным распределением весов. В случае L^1-регуляризации штраф $\alpha\Omega(\boldsymbol{w}) = \alpha\Sigma_i|w_i|$, применяемый для регуляризации функции стоимости, эквивалентен члену, содержащему логарифм априорного распределения, который максимизируется байесовским выводом на основе MAP, когда в качестве априорного используется изотропное распределение Лапласа (3.26) векторов $\boldsymbol{w} \in \mathbb{R}^n$:

$$\log p(\boldsymbol{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha \|\boldsymbol{w}\|_1 + n\log\alpha - n\log 2. \tag{7.24}$$

С точки зрения обучения посредством максимизации относительно $\boldsymbol{w}$, мы можем игнорировать члены $\log\alpha - \log 2$, поскольку они зависят от $\boldsymbol{w}$.

7.2. Штраф по норме как оптимизация с ограничениями

Рассмотрим функцию стоимости, регуляризированную путем добавления штрафа по норме параметров:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}). \tag{7.25}$$

Напомним (см. раздел 4.4), что для минимизации функции при наличии ограничений мы можем построить обобщенную функцию Лагранжа, состоящую из исходной целевой функции плюс набор штрафов. Каждый штраф имеет вид произведения коэффициента, называемого множителем Каруша–Куна–Таккера (ККТ), на функцию, показывающую, удовлетворяется ли ограничение. Если мы хотим, чтобы ограничение $\Omega(\boldsymbol{\theta})$ было меньше некоторой константы k, то можем определить такую обобщенную функцию Лагранжа:

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k). \tag{7.26}$$

Решение задачи с ограничениями имеет вид:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha). \tag{7.27}$$

Как описано в разделе 4.4, для решения этой задачи необходимо модифицировать и $\boldsymbol{\theta}$, и α. В разделе 4.5 был приведен пример линейной регрессии с L^2-ограничением. Есть также много других процедур – в одних используется метод градиентного спуска, в других аналитически ищутся точки, в которых градиент равен нулю, – но в любом случае α должно увеличиваться, когда $\Omega(\boldsymbol{\theta}) > k$, и уменьшаться, когда $\Omega(\boldsymbol{\theta}) < k$. Любое положительное α побуждает $\Omega(\boldsymbol{\theta})$ к уменьшению. Оптимальное значение α^* поощряет уменьшение $\Omega(\boldsymbol{\theta})$, но не настолько сильно, чтобы $\Omega(\boldsymbol{\theta})$ оказалось меньше k.

Чтобы составить представление о влиянии ограничения, зафиксируем α^* и будем рассматривать задачу как функцию только от $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha^*\Omega(\boldsymbol{\theta}). \tag{7.28}$$

Это в точности то же самое, что регуляризированная задача обучения путем минимизации $\tilde{J}$. Таким образом, можно считать, что штраф по норме параметра налагает ограничение на веса. Если в качестве нормы Ω берется L^2, то веса должны лежать

в единичном L^2-шаре. Если же Ω – это норма L^1, то веса должны лежать в области с ограниченной нормой L^1. Обычно нам неизвестен размер области ограничений, соответствующей снижению весов с коэффициентом α^*, потому что значение α^* не говорит прямо о значении k. В принципе, можно решать уравнение относительно k, но связь между k и α^* зависит от вида J. Но хотя мы не знаем точный размер области ограничений, мы можем грубо управлять им, уменьшая или увеличивая α с целью увеличить или уменьшить область ограничений. Чем больше α, тем меньше область ограничений, и наоборот.

Иногда мы хотим использовать явные ограничения, а не штрафы. Как описано в разделе 4.4, мы можем модифицировать алгоритм, например стохастического градиентного спуска, так чтобы он производил шаг спуска по $J(\boldsymbol{\theta})$, а затем выполнял обратное проецирование $\boldsymbol{\theta}$ в ближайшую точку, для которой $\Omega(\boldsymbol{\theta}) < k$. Это бывает полезно, если мы заранее знаем, какое значение k приемлемо, и не хотим тратить времени на поиск значения α, соответствующего этому k.

Еще одна причина использовать явные ограничения и обратное проецирование, а не косвенные ограничения в виде штрафов, состоит в том, что из-за штрафа процедура невыпуклой оптимизации может застрять в локальном минимуме, соответствующем малому $\boldsymbol{\theta}$. При обучении нейронных сетей это обычно проявляется в виде сети, обучившейся с несколькими «мертвыми блоками». Так называются блоки, которые мало что вносят в поведение обученной сетью функции, потому что веса всех входных или выходных сигналов очень малы. При обучении со штрафом на норму весов такие конфигурации могут оказаться локально оптимальными, даже если возможно значительно уменьшить J, сделав веса больше. Явные ограничения, реализованные с помощью обратного проецирования, в таких случаях могут работать гораздо лучше, потому что не поощряют приближения весов к началу координат. Такие явные ограничения вступают в силу, только когда веса становятся большими и грозят выйти за пределы области ограничений.

Наконец, явные ограничения на основе обратного проецирования могут быть полезны, потому что привносят устойчивость в процедуру оптимизации. Если скорость обучения высока, то есть риск попасть в петлю с положительной обратной связью, когда большие веса приводят к большим градиентам, а это, в свою очередь, приводит к большому обновлению весов. Если такие обновления стабильно увеличивают веса, то θ быстро отдаляется от начала координат, пока не наступит численное переполнение. Явные ограничения с обратным проецированием не дают таким петлям вызывать неограниченный рост весов. В работе Hinton et al. (2012c) рекомендуется использовать ограничения в сочетании с высокой скоростью обучения, чтобы быстрее исследовать пространство параметров без потери устойчивости.

В частности, рекомендуется стратегия, предложенная в работе Srebro and Shraibman (2005): ограничивать норму каждого столбца матрицы весов слоя нейронной сети, а не норму Фробениуса всей матрицы. Ограничение нормы каждого столбца препятствует назначению высокого веса отдельным скрытым блокам. Если преобразовать это ограничение в штраф в функции Лагранжа, то он был бы похож на снижение весов по норме L^2, только у веса каждого скрытого блока был бы отдельный множитель ККТ. Все эти множители по отдельности подвергались бы динамическому обновлению, так чтобы каждый скрытый блок подчинялся своему ограничению. На практике ограничения на нормы столбцов всегда реализуются с помощью явных ограничений с обратным проецированием.

7.3. Регуляризация и недоопределенные задачи

В некоторых случаях без регуляризации просто невозможно корректно поставить задачу машинного обучения. Многие линейные модели в машинном обучении, в т. ч. линейная регрессия и PCA, включают обращение матрицы $\boldsymbol{X}^\top\boldsymbol{X}$. Это невозможно, если $\boldsymbol{X}^\top\boldsymbol{X}$ сингулярна. Матрица может оказаться сингулярной, если порождающее распределение действительно не имеет дисперсии в некотором направлении или если в некотором направлении не наблюдается дисперсии, потому что число примеров (строк $\boldsymbol{X}$) меньше числа входных признаков (столбцов $\boldsymbol{X}$). В таком случае во многих вариантах регуляризации прибегают к обращению матрицы $\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I}$. Гарантируется, что такая регуляризированная матрица обратима.

У этих линейных задач может существовать решение в замкнутой форме, если соответствующая матрица обратима. Может также случиться, что задача, не имеющая решения в замкнутой форме, недоопределена. Примером является применение логистической регрессии к задаче, в которой классы линейно разделимы. Если вектор весов $\boldsymbol{w}$ может дать идеальную классификацию, то вектор $2\boldsymbol{w}$ также дает идеальную классификацию, но более высокое правдоподобие. Итеративная процедура оптимизации, например стохастический градиентный спуск, будет все время увеличивать абсолютную величину $\boldsymbol{w}$ и теоретически никогда не остановится. На практике рано или поздно веса достигнут такой величины, что произойдет переполнение, и дальнейшее поведение зависит от того, как программист решил обрабатывать значения, не являющиеся числами.

В большинстве вариантов регуляризации гарантируется сходимость итерационных методов, применяемых к недоопределенным задачам. Например, в случае снижения весов градиентный спуск перестанет увеличивать веса, когда угловой коэффициент правдоподобия сравняется с коэффициентом снижения весов.

Идея применения регуляризации к решению недоопределенных задач выходит за рамки одного лишь машинного обучения. Она полезна и в нескольких классических задачах линейной алгебры.

В разделе 2.9 мы видели, что недоопределенную систему линейных уравнений можно решить с помощью псевдообратной матрицы Мура–Пенроуза. Напомним ее определение:

$$\boldsymbol{X}^+ = \lim_{\alpha\searrow 0}(\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X}^\top. \tag{7.29}$$

Мы узнаём в уравнении (7.29) линейную регрессию со снижением весов. Точнее, (7.29) это предел уравнения (7.17), когда коэффициент регуляризации стремится к нулю. Поэтому псевдообращение можно интерпретировать как стабилизацию недоопределенных задач с помощью регуляризации.

7.4. Пополнение набора данных

Самый лучший способ повысить обобщаемость модели машинного обучения – обучить ее на большем объеме данных. Но, конечно же, на практике объем данных ограничен. Один из путей решения проблемы – добавить в обучающий набор искусственно сгенерированные данные. Для некоторых задач создать такие данные довольно легко.

Проще всего применить этот подход к классификации. Классификатор принимает сложный многомерный вход $\boldsymbol{x}$ и сопоставляет ему идентификатор категории y. Это

означает, что для классификатора главное – инвариантность относительно широкого спектра преобразований. Мы легко можем сгенерировать новые пары $(\boldsymbol{x}, y)$ путем различных преобразований данных $\boldsymbol{x}$, уже имеющихся в обучающем наборе.

Применение этого подхода ко многим другим задачам сталкивается с трудностями. Например, сложно сгенерировать искусственные данные для задачи оценивания плотности, пока мы не знаем ее решения.

Пополнение набора данных доказало высокую эффективность в конкретной задаче классификации: распознавание объектов. Изображения характеризуются высокой размерностью и огромным количеством факторов изменчивости, многие из которых легко смоделировать. Такие операции, как сдвиг обучающих изображений на несколько пикселей в каждом направлении, могут значительно улучшить обобщаемость, даже если при проектировании модели уже была заложена частичная инвариантность относительно параллельных переносов путем применения методов свертки и пулинга, описанных в главе 9. Многие другие операции, например поворот или масштабирование изображения, также оказались весьма эффективными.

Но не следует применять преобразования, которые могут изменить правильный класс. Например, в случае распознавания символов нужно отличать «b» от «d» и «6» от «9», поэтому в таких задачах отражение относительно вертикальной оси и поворот на 180 градусов – недопустимые способы пополнения набора данных.

Существуют также преобразования, относительно которых классификаторы должны быть инвариантными, но выполнить которые не так-то просто. Например, внеплоскостной поворот невозможно реализовать как простую геометрическую операцию над входными пикселями.

Пополнение набора данных эффективно также в задачах распознавания речи (Jaitly and Hinton, 2013).

Привнесение шума во входные данные нейронной сети (Sietsma and Dow, 1991) тоже можно рассматривать как вид пополнения данных. Для многих задач классификации и даже регрессии требуется, чтобы задача была разрешима и тогда, когда данные немного зашумлены. Однако, как оказалось, нейронные сети не очень устойчивы к шуму (Tang and Eliasmith, 2010). Один из путей повышения робастности нейронных сетей – обучать их на данных, к которым добавлен случайный шум. Привнесение шума – часть некоторых алгоритмов обучения без учителя, например шумоподавляющего автокодировщика (Vincent et al., 2008). Эта идея работает и в случае, когда шум примешивается к скрытым блокам, что можно расценивать как пополнение набора данных на нескольких уровнях абстракции. В недавней работе Poole et al. (2014) показано, что этот подход может оказаться весьма эффективным, при условии что величина шума тщательно подобрана. Прореживание, мощная стратегия регуляризации, описанная в разделе 7.12, тоже может считаться процессом конструирования новых входных данных путем *умножения* на шум.

При сравнении результатов эталонных тестов алгоритмов машинного обучения важно принимать во внимание пополнение набора данных. Зачастую вручную спроектированные схемы пополнения могут кардинально уменьшить ошибку обобщения метода машинного обучения. Для сравнения качества разных алгоритмов необходимо ставить эксперименты в контролируемых условиях. Сравнивая алгоритм A с алгоритмом B, следите за тем, чтобы оба алгоритма оценивались с применением одной и той же схемы пополнения. Предположим, что алгоритм A плохо работает на непополненном наборе данных, а алгоритм B хорошо работает, когда набор данных

пополнен различными искусственно сгенерированными примерами. В таком случае вполне вероятно, что причиной улучшения качества стали сами искусственные преобразования, а не особенности алгоритма В. Иногда для решения о том, правильно ли поставлен эксперимент, необходимо субъективное суждение. Например, алгоритмы машинного обучения, привносящие шум во входные данные, выполняют некий вид пополнения набора данных. Обычно универсальные операции (например, добавление гауссова шума) считаются частью самого алгоритма, а операции, специфичные для конкретной предметной области (например, случайная обрезка изображения), – отдельными шагами предварительной обработки.

7.5. Робастность относительно шума

В разделе 7.4 наложение шума на входные данные обосновывалось как стратегия пополнения набора данных. Для некоторых моделей добавление шума с очень малой дисперсией к входу модели эквивалентно штрафованию по норме весов (Bishop, 1995a,b). В общем случае важно помнить, что привнесение шума может дать куда больший эффект, чем простое уменьшение параметров, особенно если шум примешивается к скрытым блокам. Подача шума на скрытые блоки – тема настолько важная, что заслуживает отдельного обсуждения; алгоритм прореживания, описанный в разделе 7.12, – главное развитие этого подхода.

Еще один способ использования шума ради регуляризации моделей – прибавление его к весам. Эта техника применялась в основном в контексте рекуррентных нейронных сетей (Jim et al., 1996; Graves, 2011). Ее можно интерпретировать как стохастическую реализацию байесовского вывода весов. В байесовском подходе к обучению веса модели считаются неопределенными и представляются распределением вероятности, отражающим эту неопределенность. Прибавление шума к весам – практически удобный стохастический способ отразить неопределенность.

Применение шума к весам можно также интерпретировать как эквивалент (при определенных допущениях) более традиционной формы регуляризации, поощряющей устойчивость обучаемой функции. Рассмотрим случай регрессии, когда мы хотим обучить функцию $\hat{y}(\boldsymbol{x})$, отображающую вектор признаков $\boldsymbol{x}$, в скаляр с использованием в качестве функции стоимости среднеквадратической ошибки между предсказаниями модели $\hat{y}(\boldsymbol{x})$ и истинными значениями y:

$$J = \mathbb{E}_{p(x,y)}[\hat{y}(\boldsymbol{x}) - y)^2]. \tag{7.30}$$

Обучающий набор содержит m помеченных примеров $\{(\boldsymbol{x}^{(1)}, y^{(1)}), ..., (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

Теперь предположим, что к каждому входу добавлено случайное возмущение $\varepsilon_{\boldsymbol{W}} \sim \mathcal{N}(\boldsymbol{\varepsilon}; \mathbf{0}, \eta\boldsymbol{I})$ сетевых весов. Допустим, что мы имеем дело со стандартным l-слойным МСП. Обозначим возмущенную модель $\hat{y}_{\varepsilon_{\boldsymbol{W}}}(\boldsymbol{x})$. Несмотря на привнесенный шум, мы по-прежнему хотим минимизировать среднеквадратическую ошибку выхода сети. Таким образом, целевая функция принимает вид:

$$\tilde{J}_{\boldsymbol{W}} = \mathbb{E}_{p(\boldsymbol{x},y,\varepsilon_{\boldsymbol{W}})}[(\hat{y}_{\varepsilon_{\boldsymbol{W}}}(\boldsymbol{x}) - y)^2] \tag{7.31}$$

$$= \mathbb{E}_{p(\boldsymbol{x},y,\varepsilon_{\boldsymbol{W}})}[(\hat{y}^2_{\varepsilon_{\boldsymbol{W}}}(\boldsymbol{x}) - 2y\hat{y}_{\varepsilon_{\boldsymbol{W}}}(\boldsymbol{x}) + y^2]. \tag{7.32}$$

Для малых η минимизация J с добавленным весовым шумом (с ковариацией $\eta\boldsymbol{I}$) эквивалентна минимизации J с добавленным членом регуляризации $\eta\mathbb{E}_{p(\boldsymbol{x},y)}[\|\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x})\|^2]$.

Такая форма регуляризации поощряет сдвиг в область пространства параметров, где малые возмущения весов оказывают слабое влияние на выход. Иными словами, модель сдвигается в область нечувствительности к небольшим изменениям весов, т. е. ищутся не просто минимумы, а минимумы, окруженные плоскими участками (Hochreiter and Schmidhuber, 1995). В случае линейной регрессии (когда, например, $\hat{y}(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$) этот регуляризирующий член упрощается до $\eta \mathbb{E}_{p(\boldsymbol{x})}[\|\boldsymbol{x}\|^2]$, а эта функция не зависит от параметров и потому не дает вклада в градиент $\tilde{J}_{\boldsymbol{w}}$ относительно параметров модели.

7.5.1. Привнесение шума в выходные метки

В большинстве наборов данных выходные метки y содержат сколько-то ошибок. Максимизация $\log p(y \mid \boldsymbol{x})$, когда y ошибочно, чревата неприятными последствиями. Предотвратить это можно, в частности, путем явного моделирования шума в метках. Так, мы можем предположить, что для некоторой малой константы ε метка обучающего набора y правильна с вероятностью $1 - \varepsilon$, а в противном случае правильной может быть любая другая возможная метка. Это предположение легко включить в функцию стоимости аналитически, а не путем явной выборки зашумленных примеров. Например, **сглаживание меток** регуляризирует модель, основанную на функции softmax с k выходными значениями, путем замены жестко заданных идентификаторов классов 0 и 1 метками $\varepsilon/(k-1)$ и $1 - \varepsilon$ соответственно. Затем к этим мягким меткам можно применить стандартную функцию потерь на базе перекрестной энтропии. Обучение методом максимального правдоподобия с softmax-классификатором и жесткими метками может никогда не сойтись – softmax не умеет предсказывать вероятность, в точности равную 0 или 1, поэтому будет бесконечно продолжать обучать все большие и большие веса, делая все более экстремальные предсказания. Такого развития событий можно избежать, применив другие стратегии регуляризации, например снижение весов. У сглаживания меток есть то преимущество, что оно предотвращает стремление к жестким вероятностям, не ставя под угрозу правильность классификации. Эта стратегия используется начиная с 1980-х годов и до сих пор занимает видное место в современных нейронных сетях (Szegedy et al., 2015).

7.6. Обучение с частичным привлечением учителя

В случае обучения с частичным привлечением учителя для оценивания $P(\mathbf{y} \mid \mathbf{x})$ или предсказания $\mathbf{y}$ по $\mathbf{x}$ используются как непомеченные примеры из $P(\mathbf{x})$, так и помеченные из $P(\mathbf{x}, \mathbf{y})$.

В контексте глубокого обучения под обучением с частичным привлечением учителя обычно понимают обучение представления $\boldsymbol{h} = f(\boldsymbol{x})$. Цель – сделать так, чтобы примеры из одного класса имели похожие представления. Обучение без учителя может дать полезную информацию о том, как сгруппировать примеры в пространстве представлений. Примеры, попадающие в один кластер в пространстве входов, должны отображаться в сходные представления. Линейный классификатор в новом пространстве во многих случаях может достичь более высокого качества (Belkin and Niyogi, 2002; Chapelle et al., 2003). Проверенный временем вариант этого подхода – применить метод главных компонент в качестве шага предварительной обработки до применения классификатора (к спроецированным данным).

Вместо включения раздельных компонент с учителем и без учителя в одну модель можно построить модели, в которых порождающая модель $P(\mathbf{x})$ или $P(\mathbf{x}, \mathbf{y})$ разделяет параметры с дискриминантной моделью $P(\mathbf{y} \mid \mathbf{x})$. Затем можно выбрать компромисс между критерием обучения с учителем $-\log P(\mathbf{y} \mid \mathbf{x})$ и критерием без учителя или порождающим критерием (например, $-\log P(\mathbf{x})$ или $-\log P(\mathbf{x}, \mathbf{y})$). Тогда порождающий критерий выражает некую априорную гипотезу о решении задачи обучения с учителем (Lasserre et al., 2006), а именно что структура $P(\mathbf{x})$ связана со структурой $P(\mathbf{y} \mid \mathbf{x})$ таким способом, который улавливается совместной параметризацией. Управляя тем, какая часть порождающего критерия включается в общий, мы можем найти лучший компромисс, чем чисто порождающий или чисто дискриминантный критерий обучения (Lasserre et al., 2006; Larochelle and Bengio, 2008).

В работе Salakhutdinov and Hinton (2008) описан способ обучения ядра ядерного метода регрессии, в котором использование непомеченных примеров для моделирования $P(\mathbf{x})$ оказывается существенно лучше, чем $P(\mathbf{y} \mid \mathbf{x})$.

Дополнительные сведения об обучении с частичным привлечением учителя см. в работе Chapelle et al. (2006).

7.7. Многозадачное обучение

Многозадачное обучение (Caruana, 1993) – способ улучшить обобщаемость путем пулинга примеров (которые можно рассматривать как мягкие ограничения на параметры), возникающих в нескольких задачах. Дополнительные обучающие примеры оказывают большее давление на параметры модели, отдавая предпочтение значениям, которые хорошо обобщаются, и точно так же, если часть модели совместно используется несколькими задачами, то эта часть испытывает больше ограничений, направляющих ее в сторону хороших значений (в предположении, что разделение обосновано), что зачастую улучшает обобщаемость.

На рис. 7.2 показана широко распространенная форма многозадачного обучения, при которой разные задачи обучения с учителем (предсказание $\mathbf{y}^{(i)}$ при известном $\mathbf{x}$) разделяют один и тот же вход $\mathbf{x}$, а также некоторое промежуточное представление $\boldsymbol{h}^{(\text{shared})}$, запоминающее общий пул факторов. В общем случае модель можно разделить на части двух видов и ассоциированные с ними параметры:

1) параметры, специфичные для конкретной задачи (для достижения хорошей обобщаемости им нужны только примеры, относящиеся к «своей» задаче). Это верхние слои нейронной сети на рис. 7.2;
2) общие параметры, разделяемые всеми задачами (они извлекают выгоду из организации пула данных всех задач). Это нижние слои сети на рис. 7.2.

Улучшенной обобщаемости и ограничения ошибки обобщения (Baxter, 1995) можно достичь благодаря разделяемым параметрам, статистическую мощность которых можно значительно повысить (соразмерно с увеличенным числом примеров для разделяемых параметров, по сравнению со случаем однозадачных моделей). Разумеется, для этого нужно, чтобы выполнялись некоторые предположения относительно статистической связи между различными задачами, выражающие тот факт, что у некоторых задач действительно имеется что-то общее.

С точки зрения глубокого обучения, априорная гипотеза состоит в следующем: *среди факторов, объясняющих вариативность, наблюдаемую в данных, ассоциированных с разными задачами, некоторые являются общими для двух или более задач.*

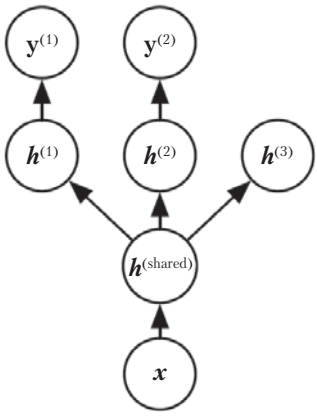

Рис. 7.2 ❖ Многозадачное обучение находит несколько применений в системах глубокого обучения, на этом рисунке показана типичная ситуация, когда задачи разделяют общий вход, но у них разные выходные случайные величины. Нижние слои глубокой сети (все равно, идет ли речь об обучении с учителем и прямом распространении или присутствует порождающая компонента с направленными вниз стрелками) могут разделяться задачами, тогда как специфичные для задач параметры (ассоциированные соответственно с весами входных и выходных сигналов $\boldsymbol{h}^{(1)}$ и $\boldsymbol{h}^{(2)}$) могут быть обучены в дополнение к тем, что входят в разделяемое представление $\boldsymbol{h}^{(\text{shared})}$. Предполагается, что существует общий пул факторов, объясняющих вариативность входа $\mathbf{x}$, а с каждой задачей ассоциированно подмножество этих факторов. В данном примере дополнительно предполагается, что блоки верхнего скрытого слоя $\boldsymbol{h}^{(1)}$ и $\boldsymbol{h}^{(2)}$ специализированы для каждой задачи (предсказывают соответственно $\mathbf{y}^{(1)}$ и $\mathbf{y}^{(2)}$), тогда как некое промежуточное представление $\boldsymbol{h}^{(\text{shared})}$ разделяется всеми задачами. В контексте обучения без учителя имеет смысл не ассоциировать с некоторыми верхнеуровневыми факторами никакие выходы задач ($\boldsymbol{h}^{(3)}$): это те факторы, которые объясняют часть вариативности входа, но не имеют отношения к предсказаниям $\mathbf{y}^{(1)}$ или $\mathbf{y}^{(2)}$

7.8. Ранняя остановка

При обучении больших моделей, репрезентативная емкость которых достаточна для переобучения, мы часто наблюдаем, что ошибка обучения монотонно убывает со временем, но ошибка на контрольном наборе снова начинает расти. Пример такого устойчивого поведения показан на рис. 7.3.

Это означает, что мы могли бы получить модель с более низкой ошибкой на контрольном наборе (и, хочется надеяться, на тестовом тоже), вернувшись к тем значениям параметров, которые существовали на момент наименьшей ошибки. Всякий раз как ошибка на контрольном наборе улучшается, мы сохраняем копию параметров модели. Когда алгоритм обучения завершается, мы возвращаем именно эти, а не самые последние параметры. А завершается алгоритм, когда на протяжении заранее заданного числа итераций не удается улучшить параметры, по сравнению с наилучшими запомненными ранее. Формально эта процедура описана в алгоритме 7.1.

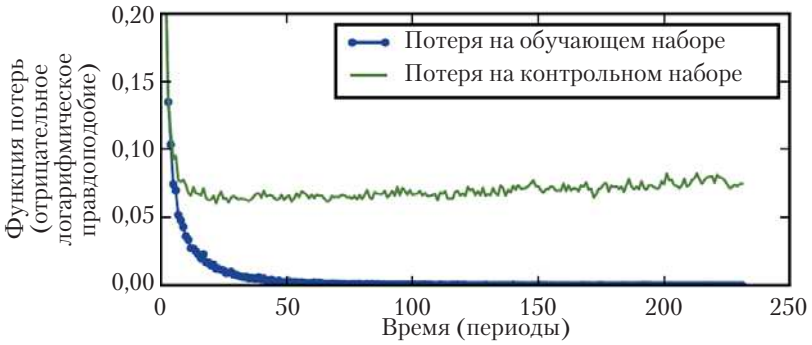

Рис. 7.3 ❖ Кривые обучения, показывающие изменение отрицательного логарифмического правдоподобия со временем (представленного в виде количества итераций обучения на наборе данных, или периодов). В данном случае обучалась maxout-сеть на наборе MNIST. Отметим, что целевая функция монотонно убывает, но средняя потеря на контрольном наборе в какой-то момент снова начинает расти, так что образуется U-образная кривая

Эта стратегия называется **ранней остановкой**. Пожалуй, это самая распространенная форма регуляризации в машинном обучении. Своей популярностью она обязана простоте и эффективности.

Алгоритм 7.1. Метаалгоритм ранней остановки для определения оптимального времени обучения. Эта общая стратегия хорошо работает для широкого круга алгоритмов обучения и способов оценки ошибки на контрольном наборе.

Обозначим n число шагов между вычислениями ошибки.
Обозначим p «терпение» – сколько раз разрешается наблюдать ухудшение ошибки на контрольном наборе, прежде чем отказаться от продолжения.
Обозначим $\boldsymbol{\theta}_o$ начальные параметры.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Обновить $\boldsymbol{\theta}$, дав алгоритму обучения проработать n шагов.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$

end if
end while
Наилучшие параметры $\boldsymbol{\theta}^*$, наилучшее число шагов обучения i^*.

Одна из возможных интерпретаций ранней остановки – очень эффективный алгоритм выбора гиперпараметров. С этой точки зрения, число шагов обучения – просто еще один гиперпараметр. На рис. 7.3 видно, что кривая качества этого гиперпараметра, измеренного на контрольном наборе, имеет U-образную форму. Большинство гиперпараметров, управляющих емкостью модели, имеет именно такую кривую качества, как было продемонстрировано на рис. 5.3. В случае ранней остановки мы управляем эффективной емкостью модели, определяя, сколько шагов ей может потребоваться для аппроксимации обучающего набора. Большинство гиперпараметров приходится выбирать, применяя дорогостоящий процесс выдвижения и проверки гипотез: вначале мы задаем гиперпараметр, а потом производим несколько шагов обучения, чтобы посмотреть, что получилось. Гиперпараметр «время обучения» уникален в том, что по определению в одном прогоне цикла обучения проверяется сразу много его значений. Когда этот параметр автоматически устанавливается путем ранней остановки, платить приходится только за периодическую проверку на контрольном наборе в ходе обучения. В идеале это следует делать параллельно с процессом обучения на отдельной машине, отдельном процессоре или отдельном GPU, не задействованных в основном процессе обучения. Если таких ресурсов нет, то стоимость периодических вычислений можно уменьшить: сделать контрольный набор небольшим, по сравнению с обучающим, или вычислять ошибку на контрольном наборе не так часто, смирившись с меньшей разрешающей способностью оценки оптимального времени обучения.

К дополнительным расходам на раннюю остановку следует также отнести хранение копии наилучших параметров. Вообще говоря, эти расходы пренебрежимо малы, поскольку параметры можно хранить в медленной памяти большого объема (например, обучение производится в памяти GPU, а оптимальные параметры хранятся в памяти хост-компьютера или на диске). Поскольку оптимальные параметры записываются сравнительно редко и никогда не читаются в процессе обучения, такие нечастые операции записи слабо сказываются на общем времени обучения.

Ранняя остановка – ненавязчивая форма регуляризации в том смысле, что не требуется вносить почти никаких изменений в базовую процедуру обучения, целевую функцию или множество допустимых значений параметров. Следовательно, раннюю остановку можно легко использовать, не изменяя динамику обучения. Совершенно не так обстоит дело со снижением весов, когда нужно внимательно следить за тем, чтобы не снизить веса слишком сильно и не завести сеть в плохой локальный минимум, соответствующий патологически малым весам.

Раннюю остановку можно использовать автономно или в сочетании с другими стратегиями регуляризации. Даже если применяются стратегии регуляризации, модифицирующие целевую функцию во имя лучшей обобщаемости, редко бывает так, что наилучшая обобщаемость достигается в локальном минимуме целевой функции.

Для ранней остановки необходим контрольный набор, а значит, часть обучающих данных не следует подавать на вход модели. Чтобы использовать эти отложенные данные более эффективно, можно провести дополнительное обучение, после того как

начальная фаза с ранней остановкой завершилась, и тогда уже включить все обучающие данные. На этом втором раунде обучения применяются две основные стратегии.

Первая (алгоритм 7.2) – снова инициализировать модель и заново обучить ее на всех данных. Но при этом мы ограничиваемся числом шагов, которое было признано оптимальным в первом раунде. Тут есть некоторые тонкости. Например, не существует хорошего способа решить, следует ли при повторном обучении производить столько же обновлений параметров или столько же проходов по набору данных, как в первом раунде. Во втором раунде каждый проход по набору данных приводит к большему числу обновлений параметров, потому что сам обучающий набор больше.

Алгоритм 7.2. Метаалгоритм использования ранней остановки для определения времени обучения с последующим повторным обучением на всех данных

Обозначим $\boldsymbol{X}^{(\text{train})}$ и $\boldsymbol{y}^{(\text{train})}$ обучающий набор.
Разделить $\boldsymbol{X}^{(\text{train})}$ и $\boldsymbol{y}^{(\text{train})}$ на $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ и $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ соответственно.
Выполнить обучение с ранней остановкой (алгоритм 7.1), начав со случайных параметров $\boldsymbol{\theta}$ и используя $\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})}$ в качестве обучающих данных, а $\boldsymbol{X}^{(\text{valid})}$ и $\boldsymbol{y}^{(\text{valid})}$ в качестве контрольных. В результате возвращается оптимальное число шагов i^*.
Снова присвоить $\boldsymbol{\theta}$ случайные значения.
Выполнить i^* шагов обучения на наборе $\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})}$.

Другая стратегия использования всех данных – оставить параметры, полученные на первом раунде, и продолжить обучение, но уже на всех данных. Теперь у нас больше нет подсказок, после скольких шагов остановиться. Вместо этого мы следим за функцией потерь на контрольном наборе и продолжаем обучение, пока ее среднее значение не окажется меньше величины целевой функции, при которой сработала ранняя остановка. Эта стратегия позволяет избежать дорогостоящего повторного обучения модели с нуля, но ее поведение оставляет желать лучшего. Например, целевая функция может никогда не достичь нужного значения на контрольном наборе, и тогда обучение не завершится. Формально эта стратегия описана в алгоритме 7.3.

Ранняя остановка полезна также тем, что уменьшает вычислительную стоимость процедуры обучения. Помимо очевидного сокращения стоимости вследствие ограничения числа итераций, она еще и обеспечивает преимущества регуляризации, не требуя включения дополнительных штрафов в функцию стоимости и вычисления градиентов этих добавочных членов.

Каким образом ранняя остановка выступает в роли регуляризатора. Мы уже не раз отмечали, что ранняя остановка является стратегией регуляризации, но в обоснование этого заявления привели только кривые обучения, на которых ошибка на контрольном наборе имеет U-образную форму. А каков истинный механизм регуляризации модели с помощью ранней остановки?

Алгоритм 7.3. Метаалгоритм использования ранней остановки для определения того, при каком значении целевой функции начинается переобучение, с последующим продолжением обучения до тех пор, пока не будет достигнуто это значение

Обозначим $\boldsymbol{X}^{(\text{train})}$ и $\boldsymbol{y}^{(\text{train})}$ обучающий набор.
Разделить $\boldsymbol{X}^{(\text{train})}$ и $\boldsymbol{y}^{(\text{train})}$ на $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ и $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ соответственно.
Выполнить обучение с ранней остановкой (алгоритм 7.1), начав со случайных параметров $\boldsymbol{\theta}$ и используя $\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})}$ в качестве обучающих данных, а $\boldsymbol{X}^{(\text{valid})}$ и $\boldsymbol{y}^{(\text{valid})}$ в качестве контрольных. При этом обновляется $\boldsymbol{\theta}$.
$\varepsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \varepsilon$ **do**
    Выполнить n шагов обучения на наборе $\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})}$.
end while

В работах Bishop (1995a) и Sjöberg and Ljung (1995) утверждается, что благодаря ранней остановке процедура оптимизации ограничивается просмотром сравнительно небольшой области пространства параметров в окрестности начального значения параметров $\boldsymbol{\theta}_o$, как показано на рис. 7.4. Точнее, допустим, что выбрано τ шагов оптимизации (что соответствует τ итерациям обучения) и скорость обучения ε. Произведение $\varepsilon\tau$ можно рассматривать как меру эффективной емкости. В предположении, что градиент ограничен, наложение ограничений на число итераций и скорость обучения лимитируют область пространства параметров, достижимую из $\boldsymbol{\theta}_o$. В этом смысле $\varepsilon\tau$ ведет себя как величина, обратная коэффициенту снижения весов.

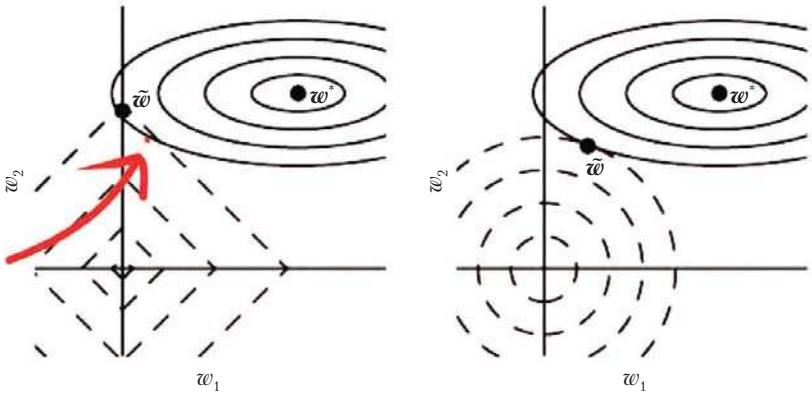

Рис. 7.4 ❖ Результат ранней остановки. (*Слева*) Сплошными линиями показаны изолинии отрицательного логарифмического правдоподобия, штриховыми – траектория стохастического градиентного спуска, начатого из начала координат. Благодаря ранней остановке траектория заканчивается не в точке $\boldsymbol{w}^*$, минимизирующей стоимость, а в более ранней точке $\tilde{\boldsymbol{w}}$. (*Справа*) Результат L^2-регуляризации для сравнения. Штриховыми кругами показаны изолинии штрафа по норме L^2, благодаря которому минимум полной функции стоимости оказывается ближе к началу координат, чем минимум нерегуляризированной функции стоимости

Действительно, можно показать, что в случае простой линейной модели с квадратичной функцией ошибки и обычного градиентного спуска ранняя остановка эквивалентна L^2-регуляризации.

Для сравнения с классической L^2-регуляризацией рассмотрим простую конфигурацию, в которой единственными параметрами являются линейные веса ($\boldsymbol{\theta} = \boldsymbol{w}$).

Функцию стоимости J можно смоделировать с помощью квадратичной аппроксимации в окрестности эмпирического оптимума весов $\boldsymbol{w}^*$:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + {}^1\!/_2(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.33}$$

где $\boldsymbol{H}$ – гессиан J относительно $\boldsymbol{w}$, вычисленный в точке $\boldsymbol{w}^*$. Поскольку мы предположили, что $\boldsymbol{w}^*$ – точка минимума $J(\boldsymbol{w})$, то $\boldsymbol{H}$ является положительно полуопределенной. Аппроксимируя разложением в ряд Тейлора, получаем выражение для градиента:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.34}$$

Мы изучим траекторию вектора параметров в процессе обучения. Для простоты пусть начальный вектор параметров совпадает с началом координат[1], $\boldsymbol{w}^{(0)} = 0$. Мы составим приближенное представление о поведении градиентного спуска по J, проанализировав градиентный спуск по $\hat{J}$:

$$\boldsymbol{w}^{(\tau)} = \boldsymbol{w}^{(\tau-1)} - \varepsilon \nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}^{(\tau-1)}) \tag{7.35}$$

$$= \boldsymbol{w}^{(\tau-1)} - \varepsilon \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.36}$$

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \varepsilon \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). \tag{7.37}$$

Перепишем это выражение в пространстве собственных векторов $\boldsymbol{H}$, воспользовавшись спектральным разложением $\boldsymbol{H}$: $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$, где $\boldsymbol{\Lambda}$ – диагональная матрица, а $\boldsymbol{Q}$ – ортогональная матрица собственных векторов.

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \varepsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*), \tag{7.38}$$

$$\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*) = (\boldsymbol{I} - \varepsilon \boldsymbol{\Lambda})\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). \tag{7.39}$$

В предположении, что $\boldsymbol{w}^{(0)} = 0$ и что ε достаточно мало, чтобы выполнялось условие $|1 - \varepsilon\lambda_i| < 1$, траектория параметров в процессе обучения после τ обновлений параметров описывается уравнением:

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = [(\boldsymbol{I} - (\boldsymbol{I} - \varepsilon \boldsymbol{\Lambda})^\tau]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.40}$$

Выражение $\boldsymbol{Q}^\top \tilde{\boldsymbol{w}}$ в уравнении (7.13) L^2-регуляризации можно переписать в виде:

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \boldsymbol{\Lambda} \boldsymbol{Q}^\top \boldsymbol{w}^*, \tag{7.41}$$

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = [\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\alpha] \boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.42}$$

Сравнивая уравнения (7.40) и (7.42), мы заключаем, что если выбрать гиперпараметры ε, α и τ, так чтобы

$$(\boldsymbol{I} - \varepsilon \boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\alpha, \tag{7.43}$$

то L^2-регуляризацию и раннюю остановку можно считать эквивалентными (по крайней мере, в предположении о квадратичной аппроксимации целевой функции). Мы можем пойти даже дальше: прологарифмировав и воспользовавшись разложением в ряд функции $\log(1 + x)$, приходим к выводу, что если все λ_i малы (то есть $\varepsilon\lambda_i \ll 1$ и $\lambda_i/\alpha \ll 1$), то

---

[1] В случае нейронных сетей мы хотим нарушить симметрию между скрытыми блоками и потому не можем инициализировать все параметры нулями (см. раздел 6.2). Однако то же рассуждение проходит и для любого другого начального значения $\boldsymbol{w}^{(0)}$.

$$\tau \approx \frac{1}{\varepsilon\alpha}, \tag{7.44}$$

$$\alpha \approx \frac{1}{\tau\varepsilon}. \tag{7.45}$$

Таким образом, в этих предположениях число итераций обучения τ играет роль величины, обратно пропорциональной параметру L^2-регуляризации, а число, обратное $\varepsilon\tau$, – роль коэффициента снижения весов.

Значения параметров, соответствующие направлениям сильной кривизны целевой функции, регуляризируются меньше, чем в направлениях меньшей кривизны. В контексте ранней остановки это в действительности означает, что параметры, соответствующие направлениям сильной кривизны, обучаются раньше параметров, соответствующих направлениям меньшей кривизны.

Выкладки, приведенные в этом разделе, показывают, что траектория длины τ обрывается в точке, соответствующей минимуму L^2-регуляризированной целевой функции. Конечно, ранняя остановка – больше, чем простое ограничение на длину траектории; ранняя остановка обычно подразумевает наблюдение за ошибкой на контрольном наборе, чтобы оборвать траекторию в удачной точке пространства. Поэтому, по сравнению со снижением весов, у ранней остановки есть то преимущество, что она автоматически определяет правильную степень регуляризации, тогда как при использовании снижения весов требуется много экспериментов с разными значениями гиперпараметра.

7.9. Связывание и разделение параметров

До сих пор в этой главе, обсуждая ограничения и штрафы, налагаемые на параметры, мы всегда отталкивались от фиксированной области или точки. Например, L^2-регуляризация (или снижение весов) штрафует параметры модели за отклонение от фиксированного значения – нуля. Но иногда требуются другие способы выражения априорных знаний о подходящих значениях параметров модели. Возможно, мы не знаем точно, какие значения должны принимать параметры, но имеющиеся знания о предметной области и архитектуре модели позволяют заключить, что между параметрами должны существовать некие зависимости.

Распространенный тип зависимости – близость некоторых параметров друг к другу. Рассмотрим такую ситуацию: есть две модели, решающие одну и ту же задачу классификации (с одинаковым набором классов), но с различающимися распределениями входных данных. Формально имеется модель A с параметрами $\boldsymbol{w}^{(A)}$ и модель B с параметрами $\boldsymbol{w}^{(B)}$. Обе модели отображают входы в разные, но взаимосвязанные выходы: $\hat{\boldsymbol{y}}^{(A)} = f(\boldsymbol{w}^{(A)}, \boldsymbol{x})$ и $\hat{\boldsymbol{y}}^{(B)} = g(\boldsymbol{w}^{(B)}, \boldsymbol{x})$.

Допустим, что задачи похожи настолько (быть может, имеют похожие распределения входов и выходов), что есть основания полагать, что их параметры должны быть близки: $\forall i\ w_i^{(A)}$ должно быть близко к $w_i^{(B)}$. Эту информацию можно применить для регуляризации: использовать штраф по норме параметров вида $\Omega(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}) = \|\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}\|_2^2$. В данном случае для вычисления штрафа мы применили норму L^2, но возможны и другие варианты.

Такой подход предложен в работе Lasserre et al. (2006), где параметры одной модели, обученной для классификации с учителем, регуляризируются с целью сделать их

близкими параметрам другой модели, обученной без учителя (для нахождения распределения наблюдаемых входных данных). Архитектуры были устроены так, чтобы многим параметрам в модели классификации можно было сопоставить параметры второй модели.

Хотя штраф по норме параметров можно использовать для регуляризации параметров с целью обеспечить их близость, более популярен другой способ: наложить ограничения, требующие, чтобы *множества параметров совпадали*. Этот метод регуляризации часто называют **разделением параметров**, поскольку мы считаем, что разные модели или компоненты моделей разделяют некоторое общее множество параметров. Важным преимуществом разделения параметров над регуляризацией с целью обеспечения близости (посредством штрафа по норме) является тот факт, что в памяти нужно хранить только подмножество всех параметров (общее множество). В некоторых моделях, например в сверточных нейронных сетях, это помогает существенно уменьшить потребность модели в памяти.

7.9.1. Сверточные нейронные сети

Конечно, самое популярное и важное применение идея разделения параметров находит в **сверточных нейронных сетях** (CHC) в компьютерном зрении.

У естественных изображений есть много статистических свойств, инвариантных к параллельному переносу. Например, фотография кошки остается таковой, если сдвинуть ее на один пиксель вправо. В CHC это свойство учитывается с помощью разделения параметров по нескольким областям изображения. Один и тот же признак (скрытый блок с одинаковыми весами) вычисляется по нескольким участкам входных данных. Это означает, что один и тот же детектор кошек найдет кошку вне зависимости от того, находится она в столбце изображения с номером i или $i + 1$.

Разделение параметров дает CHC возможность значительно уменьшить число уникальных параметров модели и увеличить размер сети, не требуя соответственного увеличения объема обучающих данных. Это по сей день остается одним из лучших примеров эффективного включения знаний о предметной области в архитектуру сети.

Более подробно CHC рассматриваются в главе 9.

7.10. Разреженные представления

Снижение весов работает благодаря штрафованию непосредственно параметров модели. Другая стратегия – штрафовать за активацию блоков нейронной сети, стремясь к разреженности активаций. Это налагает косвенный штраф на параметры модели.

Мы уже обсуждали (см. раздел 7.1.2), что штраф по норме L^1 ведет к разреженной параметризации, т. е. многие параметры обращаются в 0 (или оказываются близки к 0). С другой стороны, под разреженным понимается такое представление, многие элементы которого равны 0 (или близки к 0). Упрощенно это различие можно проиллюстрировать на примере линейной регрессии:

$$\begin{bmatrix}18\\5\\15\\-9\\-3\end{bmatrix}=\begin{bmatrix}4&0&0&-2&0&0\\0&0&-1&0&3&0\\0&5&0&0&0&0\\1&0&0&-1&0&-4\\1&0&0&0&-5&0\end{bmatrix}\begin{bmatrix}2\\3\\-2\\-5\\1\\4\end{bmatrix} \tag{7.46}$$

$\boldsymbol{y}\in\mathbb{R}^m$ $\boldsymbol{A}\in\mathbb{R}^{m\times n}$ $\boldsymbol{x}\in\mathbb{R}^n$

$$\begin{bmatrix}-14\\1\\19\\2\\23\end{bmatrix}=\begin{bmatrix}3&-1&2&-5&4&1\\4&2&-3&-1&1&3\\-1&5&4&2&-3&-2\\3&1&2&-3&0&-3\\-5&4&-2&2&-5&-1\end{bmatrix}\begin{bmatrix}0\\2\\0\\0\\-3\\0\end{bmatrix} \tag{7.47}$$

$\boldsymbol{y}\in\mathbb{R}^m$ $\boldsymbol{B}\in\mathbb{R}^{m\times n}$ $\boldsymbol{h}\in\mathbb{R}^n$

В первом случае мы имеем пример модели линейной регрессии с разреженной параметризацией, а во втором – линейную регрессию с разреженным представлением $\boldsymbol{h}$ данных $\boldsymbol{x}$. Это означает, что $\boldsymbol{h}$ – функция от $\boldsymbol{x}$, которая в каком-то смысле представляет информацию, присутствующую в $\boldsymbol{x}$, но с помощью разреженного вектора.

Для регуляризации представлений применяются те же механизмы, что для регуляризации параметров.

Регуляризация представления путем штрафа по норме производится путем прибавления к функции потерь J штрафа по норме *представления*. Этот штраф обозначается $\Omega(\boldsymbol{h})$. Как и раньше, будем обозначать регуляризированную функцию потерь $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y})=J(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y})+\alpha\Omega(\boldsymbol{h}), \tag{7.48}$$

где $\alpha\in[0,\infty)$ – относительный вес штрафа; чем больше α, тем сильнее регуляризация.

Точно так же, как штраф по норме L^1 параметров индуцирует разреженность параметров, штраф по норме L^1 элементов представления индуцирует разреженность представления: $\Omega(\boldsymbol{h})=\|\boldsymbol{h}\|_1=\Sigma_i|h_i|$. Разумеется, L^1-штраф – лишь один из возможных штрафов, приводящих к разреженному представлению. Из других назовем штрафы на основе априорного t-распределения Стьюдента (Olshausen and Field, 1996; Bergstra, 2011) и расхождения Кульбака–Лейблера (Larochelle and Bengio, 2008), особенно полезные для представлений, элементы которых принадлежат единичному интервалу. В работах Lee et al. (2008) и Goodfellow et al. (2009) приведены примеры стратегий, основанных на регуляризации, требующей, чтобы активация, усредненная по нескольким примерам $(1/m)\Sigma_i\boldsymbol{h}^i$, была близка к некоторому целевому значению, скажем, вектору, все элементы которого равны 0,01.

В других подходах разреженность представления достигается за счет жесткого ограничения на значения активации. Например, в методе **ортогонального согласованного преследования** (orthogonal matching pursuit – OMP) (Pati et al., 1993) вход $\boldsymbol{x}$ кодируется с помощью представления h, решающего следующую задачу оптимизации с ограничениями:

$$\arg\min_{\boldsymbol{h}, \|\boldsymbol{h}\|_0 < k} \|\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}\|^2, \tag{7.49}$$

где $\|\boldsymbol{h}\|_0$ – число ненулевых элементов $\boldsymbol{h}$. Эту задачу можно эффективно решить, когда на матрицу $\boldsymbol{W}$ наложено ограничение ортогональности. Этот метод часто называют OMP-k, где k обозначает допустимое число ненулевых признаков. В работе Coates and Ng (2011) показано, что OMP-1 может быть чрезвычайно эффективным экстрактором признаков для глубоких архитектур.

На самом деле любую модель со скрытыми блоками можно сделать разреженной. В этой книге нам встретится много примеров разреженной регуляризации в различных контекстах.

7.11. Баггинг и другие ансамблевые методы

Баггинг (bagging, сокращение от «**bootstrap aggregating**») – это метод уменьшения ошибки обобщения путем комбинирования нескольких моделей (Breiman, 1994). Идея заключается в том, чтобы раздельно обучить разные модели, а затем организовать их голосование за результаты на тестовых примерах. Это частный случай общей стратегии машинного обучения – **усреднения моделей**. Методы, в которых эта стратегия используется, называются **ансамблевыми методами**.

Идея усреднения моделей работает, потому что разные модели обычно не делают одни и те же ошибки на тестовом наборе.

В качестве примера рассмотрим набор из k моделей регрессии. Предположим, что каждая модель делает ошибку ε_i на каждом примере, причем ошибки имеют многомерное нормальное распределение с нулевым средним, дисперсиями $\mathbb{E}[\varepsilon_i^2] = v$ и ковариациями $\mathbb{E}[\varepsilon_i \varepsilon_j] = c$. Тогда ошибка, полученная в результате усреднения предсказаний всего ансамбля моделей, равна $(1/k)\Sigma_i \varepsilon_i$. Математическое ожидание квадрата ошибки ансамблевого предиктора равно

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \varepsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\varepsilon_i^2 + \sum_{j\neq i}\varepsilon_i\varepsilon_j\right)\right], \tag{7.50}$$

$$= \frac{1}{k}v + \frac{k-1}{k}c. \tag{7.51}$$

В случае, когда ошибки идеально коррелированы, т. е. $c = v$, среднеквадратическая ошибка сводится к v, так что усреднение моделей ничем не помогает. В случае, когда ошибки вообще не коррелированы, т. е. $c = 0$, среднеквадратическая ошибка ансамбля равна всего $(1/k)v$ и, значит, линейно убывает с ростом размера ансамбля. Иными словами, в среднем ансамбль показывает качество не хуже любого из его членов, а если члены совершают независимые ошибки, то ансамбль работает значительно лучше своих членов.

Существуют разные методы конструирования ансамбля моделей. Например, члены могут быть сформированы в результате обучения моделей разного вида разными алгоритмами или с разными целевыми функциями. Баггинг – это метод, который позволяет повторно использовать один и тот же вид модели, алгоритм обучения и целевую функцию.

Точнее, баггинг подразумевает построение k разных наборов данных. В каждом наборе столько же примеров, сколько в исходном, но строятся они путем выборки с воз-

вращением из исходного набора данных. Это означает, что с высокой вероятностью в каждом наборе данных отсутствуют некоторые примеры из исходного набора и присутствует несколько дубликатов (в среднем, если размер результирующего набора равен размеру исходного, в него попадет две трети примеров из исходного набора). Затем *i*-я модель обучается на *i*-м наборе данных. Различия в составе примеров, включенных в набор, обусловливают различия между обученными моделями. На рис. 7.5 приведена иллюстрация.

Нейронные сети дают настолько широкое разнообразие решений, что усреднение моделей может оказаться выгодным, даже если все модели обучались на одном и том же наборе данных. Различия, обусловленные случайной инициализацией, случайным выбором мини-пакетов, разными гиперпараметрами и результатами недетерминированной реализации нейронной сети, зачастую достаточны, чтобы различные члены ансамбля допускали частично независимые ошибки.

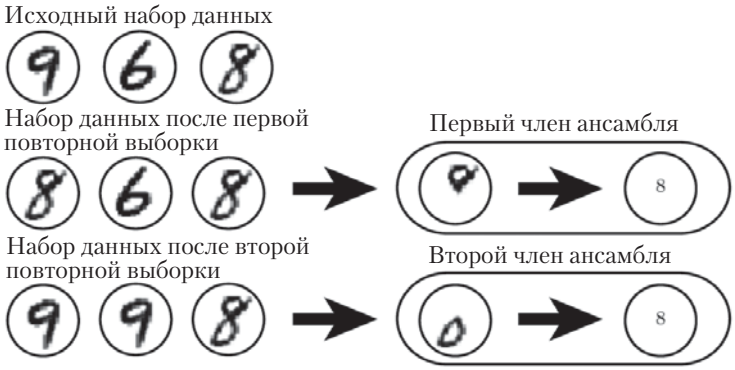

Рис. 7.5 ❖ Иллюстрация работы баггинга. Допустим, что мы обучаем детектор цифры 8 на показанном наборе данных, содержащем 8, 6 и 9. Предположим, мы хотим создать два разных набора данных на основе исходного. Процедура баггинга строит каждый набор путем выборки с возвращением из исходного набора. В первом наборе отсутствует 9, но два раза встречается 8. На этом наборе данных детектор научится, что кружочек в верхней части цифры соответствует 8. На втором наборе повторяется 9 и отсутствует 6. В этом случае детектор научится, что кружочек в нижней части соответствует 8. Каждое правило по отдельности ненадежно, но если усреднить результаты, то получится робастный детектор, дающий максимальную уверенность, только когда присутствуют оба кружочка

Усреднение моделей – исключительно мощный и надежный метод уменьшения ошибки обобщения. Его не рекомендуют использовать для эталонного тестирования алгоритмов в научных статьях, потому что любой алгоритм машинного обучения можно значительно улучшить путем усреднения моделей, но ценой увеличения времени вычислений и потребления памяти. По этой причине сравнение с эталоном обычно производят с использованием одной модели.

Соревнования по машинному обучению, как правило, выигрывают методы, производящие усреднение по десяткам моделей. Показателен недавний пример конкурса Netflix Grand Prize (Koren, 2009).

Не все методы построения ансамблей призваны сделать ансамбль более регуляризированным, чем отдельные модели. Например, метод **усиления** (boosting) (Freund and Schapire, 1996b,a) строит ансамбль с более высокой емкостью, чем у составляющих его моделей. Усиление применялось для построения ансамблей нейронных сетей (Schwenk and Bengio, 1998) путем инкрементного добавления сетей в ансамбль. Также усиление применялось для интерпретации одной нейронной сети как ансамбля (Bengio et al., 2006a) путем инкрементного добавления скрытых блоков в сеть.

7.12. Прореживание

Прореживание (dropout) (Srivastava et al., 2014) – это вычислительно недорогой, но мощный метод регуляризации широкого семейства моделей. В первом приближении прореживание можно представлять себе как метод, посредством которого баггинг становится практичным для ансамблей, состоящих из очень большого числа больших нейронных сетей. Баггинг подразумевает обучение нескольких моделей и пропускание через них каждого тестового примера. Если в качестве модели используется большая нейронная сеть, то это непрактично, потому что обучение и оценка примера обходятся дорого с точки зрения времени и памяти. Обычно используются ансамбли, содержащие от пяти до десяти сетей, – в работе Szegedy et al. (2014a), выигравшей конкурс ILSVRC, использовалось шесть – а если ансамбль больше, то работать с ним очень скоро становится неудобно. Прореживание предлагает дешевую аппроксимацию обучения и вычисления баггингового ансамбля экспоненциально большого числа нейронных сетей.

Точнее говоря, в процессе прореживания обучается ансамбль, состоящий из подсетей, получаемых удалением невыходных блоков из базовой сети, как показано на рис. 7.6. В большинстве современных нейронных сетей, основанных на последовательности аффинных преобразований и нелинейностей, можно эффективно удалить блок, умножив его выход на 0. Эта процедура требует небольшой модификации таких моделей, как сеть радиально-базисных функций, которые принимают разность между состоянием блока и некоторым опорным значением. Ниже мы для простоты опишем алгоритм прореживания в терминах умножения на 0, но путем тривиальной модификации его можно приспособить к операциям удаления блока из сети.

Напомним, что для обучения методом баггинга мы определяем k различных моделей, строим k различных наборов данных путем выборки с возвращением из обучающего набора, а затем обучаем i-ю модель на i-м наборе. Цель прореживания – аппроксимировать этот процесс на экспоненциально большом числе нейронных сетей. Точнее говоря, для обучения методом прореживания мы используем алгоритм, основанный на мини-пакетах, который делает небольшие шаги, например алгоритм стохастического градиентного спуска. При загрузке каждого примера в мини-пакет мы случайным образом генерируем битовую маску, применяемую ко всем входным и скрытым блокам сети. Элемент маски для каждого блока выбирается независимо от всех остальных. Вероятность включить в маску значение 1 (означающее, что соответствующий блок включается) – гиперпараметр, фиксируемый до начала обучения, а не функция текущего значения параметров модели или входного примера. Обычно входной блок включается с вероятностью 0.8, а скрытый – с вероятностью 0.5. Затем, как обычно, производятся прямое распространение, обратное распространение и обновление. На рис. 7.7 показано, как работает прямое распространение с прореживанием.

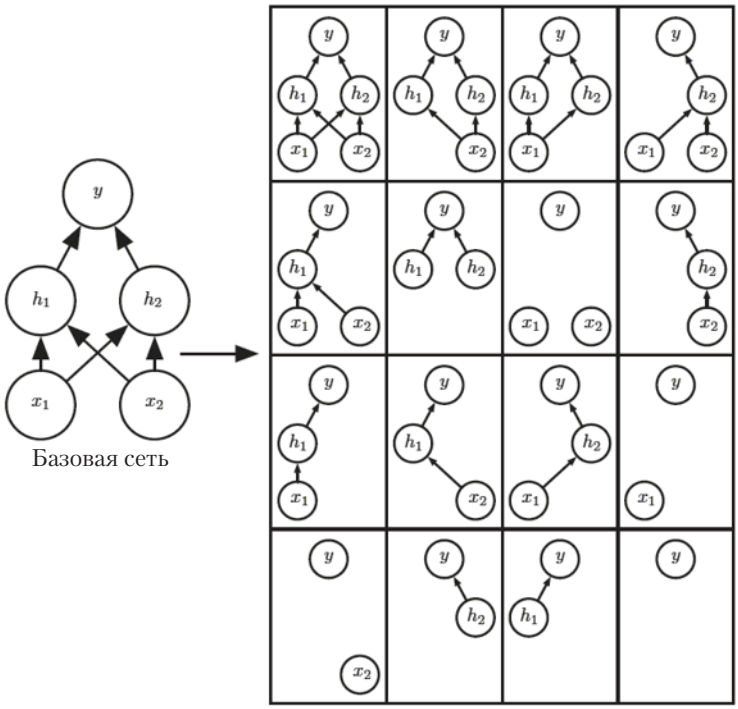

Рис. 7.6 ❖ Прореживание обучает ансамбль, состоящий из всех подсетей, которые можно построить путем удаления невыходных блоков из базовой сети. В данном случае мы начинаем с сети, имеющей два видимых и два скрытых блока. Из этих четырех блоков можно составить 16 подмножеств. На рисунке показаны все 16 подсетей, которые можно построить выбрасыванием разных подмножеств блоков из исходной сети. В этом крохотном примере есть много сетей, в которых вообще нет входных блоков или не существует пути, соединяющего вход с выходом. Эта проблема теряет остроту для сетей с более широкими слоями, когда вероятность выбросить все возможные пути от входов к выходам уменьшается

Формально, пусть вектор-маска $\boldsymbol{\mu}$ определяет, какие блоки включать, а $J(\boldsymbol{\theta}, \boldsymbol{\mu})$ – стоимость модели с параметрами $\boldsymbol{\theta}$ и маской $\boldsymbol{\mu}$. Тогда обучение с прореживанием заключается в минимизации $\mathbb{E}_{\mu} J(\boldsymbol{\theta}, \boldsymbol{\mu})$. Математическое ожидание содержит экспоненциально много членов, но мы можем получить несмещенную оценку его градиента путем выборки значений $\boldsymbol{\mu}$.

Обучение с прореживанием – не совсем то же, что баггинг. В случае баггинга все модели независимы, а в случае прореживания модели разделяют общие параметры, причем все модели наследуют разные множества параметров от родительской нейронной сети. Такое разделение параметров позволяет представить экспоненциально большое число моделей в памяти обозримого объема. В случае баггинга каждая модель обучается до сходимости на своем обучающем наборе, а в случае прореживания большинство моделей явно не обучаются вовсе – обычно исходная модель настолько велика, что для перебора всех возможных подсетей не хватило бы и вре-

мени существования вселенной. Вместо этого лишь для малой толики возможных подсетей проводится один шаг обучения, а благодаря разделению остальные подсети получают хороший набор параметров. Это и все различия. Во всем остальном прореживание повторяет алгоритм баггинга. Например, обучающий набор, который видит каждая подсеть, в действительности получен выборкой с возвращением из исходного набора.

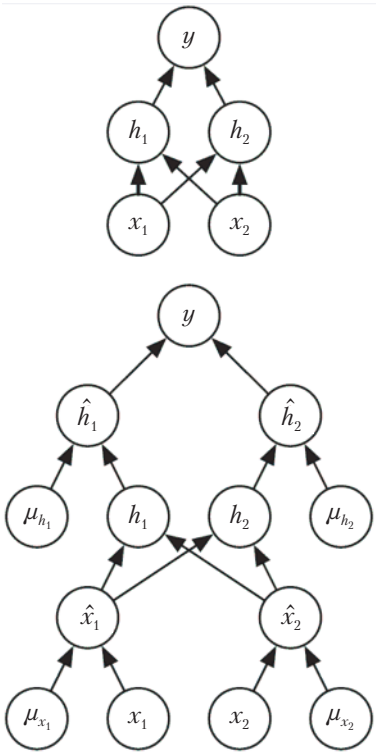

Рис. 7.7 ❖ Пример прямого распространения в сети с помощью прореживания. (*Вверху*) Сеть прямого распространения в этом примере имеет два входных блока, один скрытый слой с двумя блоками и один выходной блок. (*Внизу*) Для прямого распространения с прореживанием мы случайно выбираем вектор $\boldsymbol{\mu}$, имеющий по одному элементу для каждого входного и скрытого блока. Элементами $\boldsymbol{\mu}$ являются нули или единицы, и каждый элемент выбирается независимо от других. Вероятность единицы – гиперпараметр, который обычно равен 0.5 для скрытых блоков и 0.8 для входных. Каждый блок сети умножается на соответствующий элемент маски, а затем прямое распространение продолжается по оставшейся части сети, как обычно. Это эквивалентно случайному выбору одной из подсетей на рис. 7.6 и прямому распространению в ней

Чтобы сделать предсказания, баггинговый ансамбль должен собрать голоса всех своих членов. В этом контексте такой процесс называется **выводом**. До сих пор в описании баггинга и прореживания не было явного требования о вероятностной природе модели. Теперь мы предположим, что роль модели заключается в нахождении

распределения вероятностей. В случае баггинга i-я модель порождает распределение $p^{(i)}(y \mid \boldsymbol{x})$. Тогда предсказанием ансамбля будет среднее арифметическое всех распределений:

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x}). \tag{7.52}$$

В случае прореживания каждая подмодель, задаваемая вектором-маской $\boldsymbol{\mu}$, определяет распределение вероятности $p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$. Среднее арифметическое по всем маскам равно

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu}), \tag{7.53}$$

где $p(\boldsymbol{\mu})$ – распределение вероятности, которое использовалось для выборки $\boldsymbol{\mu}$ на этапе обучения.

Поскольку количество членов в этой сумме экспоненциально велико, вычислить ее можно только в случае, когда структура модели допускает какое-то упрощение. Пока что неизвестны нейронные сети, допускающие такое упрощение. Но мы можем аппроксимировать вывод с помощью выборки, усреднив выходы для нескольких масок. Даже 10–20 масок часто достаточно для получения хорошего качества.

Но есть подход еще лучше – он позволяет получить хорошую аппроксимацию предсказаний всего ансамбля, произведя всего одно прямое распространение. Для этого мы вместо среднего арифметического распределений, предсказанных членами ансамбля, будем вычислять среднее геометрическое. В работе Warde-Farley et al. (2014) приведены аргументы и эмпирические свидетельства в пользу того, что в этом контексте среднее геометрическое дает качество, сравнимое со средним арифметическим.

Вообще говоря, не гарантируется, что среднее геометрическое нескольких распределений вероятности само является распределением вероятности. Чтобы получить такую гарантию, мы потребуем, чтобы ни одна подмодель не назначала никаким событиям вероятность 0, а затем произведем нормировку получившегося распределения. Ненормированное распределение вероятности, определяемое средним геометрическим, имеет вид:

$$\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \boldsymbol{x}, \boldsymbol{\mu})}, \tag{7.54}$$

где d – число блоков, которые можно выбросить. Здесь для простоты взято равномерное распределение $\boldsymbol{\mu}$, но неравномерные распределения тоже допустимы. Чтобы можно было делать предсказания, необходимо перенормировать ансамбль:

$$p_{\text{ensemble}}(y \mid \boldsymbol{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \boldsymbol{x})}. \tag{7.55}$$

Основная идея (Hinton et al., 2012c) прореживания состоит в том, что p_{ensemble} можно аппроксимировать, вычислив $p(y \mid \boldsymbol{x})$ для одной модели: она включает все блоки, но веса связей, исходящих из i-го блока, умножаются на вероятность включения этого блока. Обоснование такой модификации – стремление получить правильное ожидаемое значение выхода блока. Этот подход мы называем **правилом вывода с масштабированием весов**. Пока не существует теоретического доказательства точности этого

приближенного правила вывода в глубоких нелинейных сетях, но его эмпирическое поведение очень хорошее.

Поскольку обычно принимается вероятность включения $^1/_2$, то правило масштабирования весов сводится к делению весов пополам в конце обучения, после чего модель используется как обычно. Другой способ получить тот же результат – умножать состояния блоков на 2 на этапе обучения. Как бы то ни было, цель состоит в том, чтобы ожидаемый суммарный вход в блок на этапе тестирования был приблизительно равен ожидаемому суммарному входу в тот же блок на этапе обучения, несмотря на то что в среднем половина блоков во время обучения отсутствует.

Для многих классов моделей, не имеющих нелинейных скрытых блоков, правило вывода с масштабированием весов является точным. В качестве простого примера рассмотрим регрессионный классификатор с функцией softmax и n входными переменными, представленными вектором $\mathbf{v}$:

$$P(\mathrm{y} = y \mid \mathbf{v}) = \mathrm{softmax}\left(\boldsymbol{W}^\top \mathbf{v} + \boldsymbol{b}\right)_y. \tag{7.56}$$

Мы можем проиндексировать это семейство моделей с помощью поэлементного умножения входа на двоичный вектор d:

$$P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d}) = \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v} + \boldsymbol{b}\right)_y. \tag{7.57}$$

Ансамблевый предиктор определяется путем нормировки среднего геометрического предсказаний всех членов ансамбля:

$$P_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(\mathrm{y} = y' \mid \mathbf{v})}, \tag{7.58}$$

где

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}. \tag{7.59}$$

Чтобы убедиться, что правило масштабирования весов точное, упростим $\tilde{P}_{\text{ensemble}}$:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})} \tag{7.60}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y} \tag{7.61}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \frac{\exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}{\sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_{y'}\right)}} \tag{7.62}$$

$$= \frac{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}}{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_{y'}\right)}} \tag{7.63}$$

Поскольку $\tilde{P}$ будет подвергнуто нормировке, мы можем игнорировать множители, не зависящие от y:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)} \tag{7.64}$$

$$= \exp\left(\frac{1}{2^n} \sum_{\boldsymbol{d} \in \{0,1\}^n} \boldsymbol{W}_{y,:}^{\top} (\boldsymbol{d} \odot \mathbf{v}) + b_y \right) \tag{7.65}$$

$$= \exp\left(\frac{1}{2^n} \boldsymbol{W}_{y,:}^{\top} \mathbf{v} + b_y \right). \tag{7.66}$$

Подставляя это выражение в уравнение (7.58), получаем softmax-классификатор с весами $^1/_2\boldsymbol{W}$.

Правило масштабирования весов является точным и в других конфигурациях, в т. ч. в регрессионных сетях с условно нормальными распределениями на выходе, а также в глубоких сетях, в скрытых слоях которых нет нелинейностей. Однако для глубоких моделей с нелинейностями это правило – всего лишь аппроксимация. Хотя теоретической оценки этой аппроксимации не существует, на практике она часто дает хорошие результаты. В работе Goodfellow et al. (2013a) экспериментально показано, что аппроксимация на основе масштабирования весов может работать лучше (в терминах верности классификации), чем аппроксимации методом Монте-Карло ансамблевого предиктора. Это справедливо даже тогда, когда для аппроксимации методом Монте-Карло было разрешено делать выборку из 1000 подсетей. С другой стороны, в работе Gal and Ghahramani (2015) обнаружено, что для некоторых моделей удается получить более высокую верность классификации с помощью выборки объемом 20 и аппроксимации методом Монте-Карло. Похоже, что оптимальная аппроксимация вывода зависит от задачи.

В работе Srivastava et al. (2014) показано, что прореживание эффективнее других стандартных вычислительно недорогих регуляризаторов: снижения весов, фильтрации с ограничением по норме и разреженной активации. Дальнейшего улучшения можно добиться, комбинируя прореживание с другими видами регуляризации.

Одно из преимуществ прореживания – вычислительная простота. Применение прореживания на этапе обучения требует всего $O(n)$ вычислений на каждый пример на каждое обновление – для генерирования n случайных двоичных чисел и умножения их на состояние. В зависимости от реализации может понадобиться также память объемом $O(n)$ для сохранения этих двоичных чисел до этапа обратного распространения. Стоимость вывода с помощью обученной модели в расчете на один пример такая же, как если бы прореживание не использовалось, хотя к накладным расходам следует отнести стоимость однократного деления весов на 2 до применения вывода к примерам.

У прореживания есть еще одно важное преимущество: оно не налагает существенных ограничений на тип модели или процедуру обучения. Оно одинаково хорошо работает практически с любой моделью, если в ней используется распределенное представление и ее можно обучить методом стохастического градиентного спуска. Сюда входят нейронные сети прямого распространения, вероятностные модели типа ограниченных машин Больцмана (Srivastava et al., 2014) и рекуррентные нейронные сети (Bayer and Osendorfer, 2014; Pascanu et al., 2014a). Многие другие стратегии регуляризации сравнимой мощности налагают куда более строгие ограничения на архитектуру модели.

Хотя стоимость одного шага применения прореживания к конкретной модели пренебрежимо мала, его общая стоимость для модели в целом может оказаться значительной. Будучи методом регуляризации, прореживание уменьшает эффективную

емкость модели. Чтобы компенсировать этот эффект, мы должны увеличить размер модели. Как правило, оптимальная ошибка на контрольном наборе при использовании прореживания намного ниже, но расплачиваться за это приходится гораздо большим размером модели и числом итераций алгоритма обучения. Для очень больших наборов данных регуляризация не сильно снижает ошибку обобщения. В таких случаях вычислительная стоимость прореживания и увеличение модели могут перевесить выигрыш от регуляризации.

Есть в нашем распоряжении очень мало помеченных обучающих примеров, то прореживание менее эффективно. Байесовские нейронные сети (Neal, 1996) оказываются лучше на наборе данных Alternative Splicing Dataset (Xiong et al., 2011), содержащем менее 5000 примеров (Srivastava et al., 2014). Если дополнительно имеются непомеченные данные, то отбор признаков путем обучения без учителя может превзойти прореживание.

В работе Wager et al. (2013) показано, что в случае применения к линейной регрессии прореживание эквивалентно снижению весов по норме L^2, когда для каждого входного признака задается свой коэффициент снижения веса. Абсолютная величина каждого коэффициента определяет дисперсией признака. Аналогичные результаты имеют место для других линейных моделей. Для глубоких моделей прореживание не эквивалентно снижению весов.

Использование стохастичности в обучении с прореживанием не является необходимым условием успеха. Это просто средство аппроксимации суммы по всем подмоделям. В работе Wang and Manning (2013) получены аналитические аппроксимации этой маргинализации. Найденная ими аппроксимация, известная под названием «**быстрое прореживание**», сходится быстрее благодаря уменьшению стохастичности при вычислении градиента. Этот метод можно применять и на стадии тестирования, как теоретически более обоснованную (хотя вычислительно более дорогую) аппроксимацию среднего во всем подсетям, по сравнению с масштабированием весов. Быстрое прореживание по качеству почти не уступает стандартному на небольших нейронных сетях, но пока не сумело достичь существенного улучшения и не применялось к большим задачам.

Мало того что стохастичность не является необходимой для достижения регуляризирующего эффекта прореживания, она еще и недостаточна. Чтобы продемонстрировать это, в работе Warde-Farley et al. (2014) поставлены контрольные эксперименты с помощью метода **усиленного прореживания** (dropout boosting), в котором используется точно такой же масочный шум, что в традиционном прореживании, однако эффект регуляризации не достигается. Алгоритм усиленного прореживания обучает ансамбль совместно максимизировать логарифмическое правдоподобие на обучающем наборе. В том же смысле, в каком традиционное прореживание аналогично баггингу, этот подход аналогичен усилению. Как и предполагалось, эксперименты с усиленным прореживанием не показали почти никакого эффекта регуляризации, по сравнению с обучением всей сети как одной модели. Это показывает, что интерпретация прореживания как баггинга не исчерпывается его интерпретацией как устойчивости к шуму. Эффект регуляризации баггингового ансамбля достигается, только когда стохастически выбранные члены ансамбля обучаются хорошо работать независимо друг от друга.

Идея прореживания дала начало другим стохастическим подходам к обучению экспоненциально больших ансамблей моделей, разделяющих веса. Метод DropConnect – частный случай прореживания, в котором каждое произведение одного скаляр-

ного веса и состояния одного скрытого блока рассматривается как блок, подлежащий выбрасыванию (Wan et al., 2013). Стохастический пулинг – это вариант рандомизированного пулинга (см. раздел 9.3) для построения ансамблей сверточных сетей, в котором каждая сеть занимается своей пространственной областью каждой карты признаков. До настоящего времени прореживание остается самым широко употребительным методом неявных ансамблей.

Одна из главных идей прореживания заключается в том, что обучение сети с элементами стохастичности и предсказание путем усреднения по многим стохастическим решениям является формой баггинга с разделением параметров. Выше мы описывали прореживание как баггинг ансамбля моделей, образованного путем включения и исключения блоков. Однако эта стратегия усреднения моделей может применяться не только к включению и исключению. В принципе, допустимы любые виды случайной модификации. На практике следует выбирать классы модификаций так, чтобы нейронная сеть могла обучиться стойкости к ним. В идеале хотелось бы также, чтобы семейство моделей допускало быстрое приближенное правило вывода. Можно представлять себе любой вид модификации, параметризованный вектором $\boldsymbol{\mu}$, как обучение ансамбля, состоящего из $p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{\mu})$ для всех возможных значений $\boldsymbol{\mu}$. Не требуется, чтобы число значений $\boldsymbol{\mu}$ было конечным. Например, $\boldsymbol{\mu}$ может принимать вещественные значения. В работе Srivastava et al. (2014) показано, что умножение весов на $\boldsymbol{\mu} \sim \mathcal{N}(\mathbf{1}, I)$ может по качеству превосходить прореживание, основанное на двоичных масках. Поскольку $\mathbb{E}[\boldsymbol{\mu}] = \mathbf{1}$, стандартная сеть автоматически реализует приближенный вывод в ансамбле без всякого масштабирования весов.

До сих пор мы описывали прореживание исключительно как средство эффективного приближенного баггинга. Но есть и другой, гораздо более широкий взгляд на прореживание. Метод прореживания обучает не просто баггинговый ансамбль моделей, а ансамбль моделей с общими скрытыми блоками. Это означает, что каждый скрытый блок должен демонстрировать хорошее поведение вне зависимости от того, какие еще скрытые блоки есть в модели. Скрытые блоки должны быть готовы к тому, что окажутся в другой модели. На авторов работы Hinton et al. (2012c) оказала влияние идея, заимствованная из биологии: половое размножение, состоящее в обмене генами между двумя разными организмами, оказывает давление эволюционного отбора на гены – они должны быть хороши не только сами по себе, но и готовы к обмену между организмами. Такие гены и такие признаки устойчивы к изменениям в окружающей среде, поскольку неспособны по ошибке адаптироваться к необычным признакам организма или модели. Таким образом, прореживание регуляризирует каждый скрытый блок, делая его не просто хорошим признаком, а хорошим в разных контекстах. В работе Warde-Farley et al. (2014) обучение с прореживанием сравнивается с обучением больших ансамблей, и делается вывод, что прореживание дополнительно улучшает ошибку обобщения сверх того, что может быть получено с помощью ансамблей независимых моделей.

Важно понимать, что своей мощью прореживание в немалой степени обязано тому факту, что к скрытым блокам применяется маскирующий шум. Это можно рассматривать как форму высокоразумного адаптивного уничтожения информационного содержания входа, а не уничтожения исходных входных значений. Например, если модель обучает скрытый блок h_i, который обнаруживает лицо, найдя нос, то выбрасывание h_i соответствует стиранию информации о присутствии носа в изображении. Модель должна обучить еще один h_i, который либо избыточно кодирует присутствие

носа, либо обнаруживает лицо по наличию другого признака, например рта. Традиционные приемы зашумления – добавление неструктурированного шума к входу – неспособны случайным образом стереть информацию о носе из изображения лица, разве что величина шума настолько велика, что из изображения удаляется почти вся информация. Уничтожение выделенных признаков, а не исходных значений позволяет процессу уничтожения воспользоваться всей накопленной моделью информацией о входном распределении.

Еще один важный аспект прореживания – мультипликативность шума. Если бы шум был аддитивным с фиксированным масштабом, то скрытый блок линейной ректификации h_i с добавленным шумом ε мог бы просто обучиться делать h_i очень большим, чтобы на его фоне добавленный шум казался незначительным. Мультипликативный шум препятствует таким патологическим решениям проблемы устойчивости к шуму.

Еще один алгоритм глубокого обучения, пакетная нормировка, изменяет параметризацию модели, стремясь ввести в скрытые блоки как аддитивный, так и мультипликативный шум на этапе обучения. Основная цель пакетной нормировки – улучшить оптимизацию, но шум может давать регуляризирующий эффект, так что иногда прореживание оказывается излишним. Мы вернемся к пакетной нормировке в разделе 8.7.1.

7.13. Состязательное обучение

Во многих случаях нейронные сети демонстрируют сравнимое с человеком качество в применении к независимому и одинаково распределенному тестовому набору. Поэтому естественно задаться вопросом, смогли ли эти модели достичь по-настоящему человеческого понимания таких задач. Чтобы оценить уровень понимания задачи сетью, можно поискать примеры, которые модель классифицирует неправильно. В работе Szegedy et al. (2014b) обнаружено, что даже нейронные сети, демонстрирующие верность классификации, приближающуюся к уровню человека, ошибаются почти в 100 процентах случаев, если им предъявить примеры, намеренно построенные с помощью специальной процедуры, которая ищет вход $\boldsymbol{x}'$ вблизи точки $\boldsymbol{x}$ – такой, что выход модели сильно отличается в $\boldsymbol{x}'$. Во многих случаях $\boldsymbol{x}'$ может быть настолько похожа на $\boldsymbol{x}$, что человек не сможет отличить исходный пример от состязательного, однако сеть будет давать совершенно разные предсказания. Пример показан на рис. 7.8.

Рис. 7.8 ❖ Генерация состязательного примера, примененного к сети GoogLeNet (Szegedy et al., 2014a) на наборе данных ImageNet. Прибавив к входу неощутимо малый вектор, элементы которого равны знакам элементов градиента функции стоимости, мы можем заставить GoogLeNet изменить классификацию изображения. Взято из работы Goodfellow et al. (2014b) с разрешения авторов

У состязательного обучения много применений, в частности в области компьютерной безопасности, но они выходят за рамки этой главы. Однако же они представляют интерес в контексте регуляризации, поскольку состязательное обучение позволяет уменьшить частоту ошибок на исходном независимом и одинаково распределенном тестовом наборе – путем обучения на искусственно возмущенных примерах из обучающего набора (Szegedy et al., 2014b; Goodfellow et al., 2014b). В работе Goodfellow et al. (2014b) показано, что одна из основных причин таких состязательных примеров – их избыточная линейность. Нейронные сети строятся в основном из линейных блоков. В некоторых экспериментах оказывалось, что в результате реализуемая ими полная функция почти линейна. Такие линейные функции хорошо поддаются оптимизации. К сожалению, значение линейной функции может очень быстро изменяться, если входов много. Если изменить каждый вход на ε, то линейная функция с весами $\boldsymbol{w}$ может измениться на $\varepsilon\|\boldsymbol{w}\|_1$, а в случае многомерного вектора $\boldsymbol{w}$ эта величина может оказаться очень большой. Состязательное обучение подавляет такое излишне локально чувствительное линейное поведение, побуждая сеть сохранять локальное постоянство в окрестности обучающих данных. Это можно рассматривать как явное введение априорной информации о локальном постоянстве в нейронную сеть, обучаемую с учителем.

Состязательное обучение иллюстрирует широкие возможности использования большого семейства функций в комбинации с агрессивной регуляризацией. Чисто линейные модели, например логистическая регрессия, не могут противостоять состязательным примерам, поскольку обязаны быть линейными. Нейронные сети способны представлять функции от почти линейных до почти локально постоянных и потому обладают достаточной гибкостью, чтобы уловить в обучающих данных тенденцию к линейности и вместе с тем обучиться противостоять локальным возмущениям.

Состязательные примеры предоставляют также средства для обучения с частичным привлечением учителя. Если с точкой $\boldsymbol{x}$ не ассоциирована метка, то модель сама назначает ей некоторую метку $\hat{y}$. Назначенная моделью метка может отличаться от истинной, но если модель высокого качества, то вероятность правильности $\hat{y}$ велика. Мы можем поискать состязательный пример $\boldsymbol{x}'$, который заставляет классификатор выдать метку y' – такую, что $y' \neq \hat{y}$. Состязательные примеры, при генерации которых используется не истинная метка, а метка, назначенная обученной моделью, называются **виртуальными состязательными примерами** (Miyato et al., 2015). Затем классификатор можно обучить назначать одинаковые метки примерам $\boldsymbol{x}$ и $\boldsymbol{x}'$. Это побуждает классификатор обучать функцию, устойчивую к малым изменениям вдоль многообразия, на котором лежат непомеченные данные. В основу этого подхода положено предположение о том, что разные классы обычно лежат на несвязных многообразиях, так что малое возмущение не может привести к «перепрыгиванию» с одного многообразия на другое.

7.14. Тангенциальное расстояние, алгоритм распространения по касательной и классификатор по касательной к многообразию

Многие алгоритмы машинного обучения стремятся преодолеть проклятие размерности, предполагая, что данные лежат в окрестности многообразия низкой размерности (см. раздел 5.11.3).

Одна из ранних попыток воспользоваться гипотезой о многообразии – алгоритм **тангенциального расстояния** (tangent distance) (Simard et al., 1993, 1998). Это непараметрический алгоритм ближайшего соседа, в котором в качестве метрики используется не обычное евклидово расстояние, а расстояние, выведенное из знания о многообразиях, вблизи которых сконцентрирована вероятность. Предполагается, что мы пытаемся классифицировать примеры и что примеры, принадлежащие одному и тому же многообразию, относятся к одной и той же категории. Поскольку классификатор должен быть инвариантен относительно локальных факторов вариативности, соответствующих перемещению по многообразию, имеет смысл использовать в качестве меры близости соседей $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ расстояние между многообразиями M_1 и M_2, которым эти точки принадлежат. Эта задача может оказаться вычислительно трудной (для нахождения ближайшей пары точек на M_1 и M_2 необходимо решить задачу оптимизации), но есть и дешевая альтернатива – локально аппроксимировать M_i касательной плоскостью в точке x_i и измерить расстояние между двумя касательными или между точкой и касательной плоскостью. Для этого нужно решить систему небольшого числа линейных уравнений (их число равно размерности многообразий). Разумеется, в этом алгоритме необходимо задать касательные векторы.

Похожий алгоритм **распространения по касательной** (tangent prop) (Simard et al., 1992) (рис. 7.9) обучает классификатор на основе нейронной сети с дополнительным штрафом, цель которого – сделать каждый выход $f(\boldsymbol{x})$ сети локально инвариантным относительно известных факторов вариативности. Эти факторы соответствуют перемещению вдоль многообразия, вблизи которого концентрируются примеры из одного класса. Локальная инвариантность достигается за счет требования, что вектор $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ ортогонален известным касательным векторам многообразия $\boldsymbol{v}^{(i)}$ в точке $\boldsymbol{x}$ или, эквивалентно, что производная по направлению функции f в точке $\boldsymbol{x}$ в направлениях $\boldsymbol{v}^{(i)}$ мала. Для этого добавляется регуляризирующий штраф Ω:

$$\Omega(f) = \sum_i ((\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^\top \boldsymbol{v}^{(i)})^2. \tag{7.67}$$

Конечно, этот регуляризатор можно умножить на подходящий гиперпараметр, и для большинства нейронных сетей нужно будет просуммировать по многим выходам, а не ограничиваться одним выходом $f(\boldsymbol{x})$, как мы для простоты поступили здесь. Как и в алгоритме тангенциального расстояния, касательные векторы задаются заранее, обычно исходя из формальных знаний о влиянии на изображения таких преобразований, как параллельный перенос, поворот и масштабирование. Алгоритм распространения по касательной применялся не только для обучения с учителем (Simard et al., 1992), но и в контексте обучения с подкреплением (Thrun, 1995).

Распространение по касательной тесно связано с пополнением набора данных. В обоих случаях пользователь алгоритма кодирует априорные знания о задаче, задавая множество преобразований, которые не должны изменять выход сети. Разница же в том, что в случае пополнения набора данных сеть явно обучается правильно классифицировать разные входы, созданные применением таких преобразований. А для распространения по касательной не нужно явно посещать новую входную точку. Вместо этого алгоритм аналитически регуляризирует модель, обучая ее противостоять возмущениям в направлениях, соответствующих заданному преобразованию. Хотя такой аналитический подход обладает интеллектуальной элегантностью,

у него есть два крупных недостатка. Во-первых, он регуляризирует модель только для противостояния бесконечно малым возмущениям. Явное пополнение набора данных прививает устойчивость к более сильным возмущениям. Во-вторых, инфинитезимальный подход испытывает трудности в применении к моделям, основанным на блоках линейной ректификации. В таких моделях уменьшение производных достигается только путем отключения блоков или уменьшения их весов. Они не могут уменьшить производные путем насыщения при больших значениях с сохранением больших весов, как сигмоида или гиперболический тангенс. Что же касается пополнения набора данных, то оно хорошо работает с блоками линейной ректификации, поскольку для разных преобразованных вариантов одних и тех же исходных данных могут активироваться различные подмножества ректифицированных блоков.

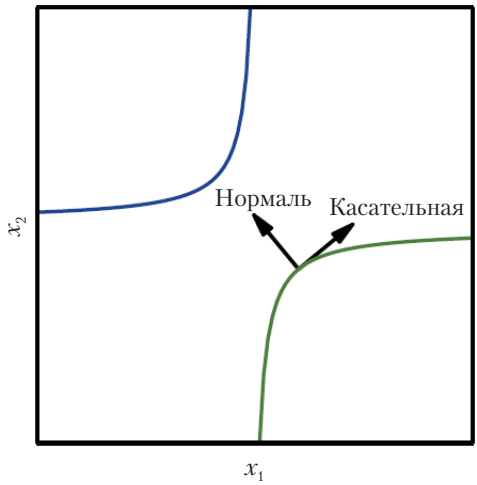

Рис. 7.9 ❖ Иллюстрация основной идеи алгоритма распространения по касательной (Simard et al., 1992) и классификатора по касательной к многообразию (Rifai et al., 2011c); тот и другой регуляризируют выходную функцию классификатора $f(\boldsymbol{x})$. Каждая кривая представляет многообразие для одного класса, в данном случае они показаны как одномерные многообразия в двумерном пространстве. На одной кривой мы выбрали точку и провели два вектора: касательный и нормальный к многообразию класса. В многомерном пространстве касательных и нормальных направлений много. Мы ожидаем, что функция классификации будет быстро изменяться в направлении нормали к многообразию и не изменяться при перемещении вдоль многообразия класса. И алгоритм распространения по касательной, и классификатор по касательной к многообразию регуляризируют $f(\boldsymbol{x})$, так чтобы она изменялась не слишком сильно, когда $\boldsymbol{x}$ перемещается вдоль многообразия. Для применения алгоритма распространения по касательной пользователь должен вручную задать функции, вычисляющие касательные направления (например, исходя из того, что малый сдвиг не изменяет класса изображения), а классификатор по касательной к многообразию оценивает касательные направления, обучая автокодировщик аппроксимировать обучающие данные. Использование автокодировщиков для оценивания многообразий рассматривается в главе 14

Распространение по касательной также связано с **двойным обратным распространением** (Drucker and LeCun, 1992) и состязательным обучением (Szegedy et al., 2014b; Goodfellow et al., 2014b). Идея двойного обратного распространения – регуляризировать якобиан с целью его уменьшения, тогда как при состязательном обучении ищутся входы рядом с исходными входами, и модель обучается так, чтобы порождать на них те же самые выходы, что на исходных входах. И распространение по касательной, и пополнение набора данных с заданными вручную преобразованиями требуют, чтобы модель была инвариантна относительно заданных направлений изменения входных данных. И двойное обратное распространение, и состязательное обучение требуют, чтобы модель была инвариантна во всех направлениях изменения входных данных при условии, что изменение мало. Как пополнение набора данных является неинфинитезимальным вариантом распространения по касательной, так состязательное обучение – неинфинитезимальный вариант двойного обратного распространения.

Классификатор по касательной к многообразию (Rifai et al., 2011c) позволяет обойтись без априорного знания касательных векторов. В главе 14 мы увидим, что автокодировщики умеют оценивать касательные векторы многообразия. Классификатор использует эту технику, чтобы избежать задания касательных векторов пользователем. На рис. 14.10 показано, что оценки касательных векторов выходят за рамки классических инвариантов, вытекающих из геометрии изображений (относительно параллельного переноса, поворота и масштабирования), и включает факторы, которые можно найти только в процессе обучения, потому что они зависят от объекта (например, движущиеся части тела). Поэтому алгоритм классификатора по касательной к многообразию прост: (1) воспользоваться автокодировщиком, чтобы выявить структуру многообразия в процессе обучения без учителя, и (2) использовать найденные касательные векторы для регуляризации классификатора на базе нейронной сети, как в алгоритме распространения по касательной (уравнение 7.67).

В этой главе мы описали большинство общих стратегий регуляризации нейронных сетей. Регуляризация – центральная тема машинного обучения, и потому мы часто будем возвращаться к ней в других главах. Еще одна важная тема – оптимизация – будет рассмотрена в следующей главе.

Глава 8

Оптимизация в обучении глубоких моделей

Алгоритмы глубокого обучения включают оптимизацию в самых разных контекстах. Например, для выполнения вывода в таких моделях, как метод главных компонент, необходимо решать задачу оптимизации. Мы часто применяем аналитическую оптимизацию для доказательства правильности или проектирования алгоритмов. Из всех многочисленных задач оптимизации, решаемых в глубоком обучении, самые трудные возникают при обучении нейронной сети. Нередко приходится затрачивать от нескольких дней до нескольких месяцев работы на сотнях машин, чтобы решить всего одну задачу обучения нейронной сети. Поскольку проблема так важна, а ее решение обходится так дорого, разработаны специальные методы оптимизации. В этой главе мы рассмотрим методы оптимизации для обучения нейронных сетей.

Если вы незнакомы с базовыми принципами градиентной оптимизации, рекомендуем прочитать главу 4, где приведен краткий обзор числовой оптимизации вообще.

В этой главе нас будет интересовать один частный случай: нахождение параметров $\boldsymbol{\theta}$ нейронной сети, значительно уменьшающих функцию стоимости $J(\boldsymbol{\theta})$, которая обычно служит мерой качества, вычисляется на всем обучающем наборе и содержит дополнительные регуляризирующие члены.

Начнем с того, чем оптимизация, используемая в алгоритме машинного обучения, отличается от чистой оптимизации. Затем перечислим несколько конкретных проблем, делающих оптимизацию нейронных сетей такой трудной задачей. После этого мы опишем несколько практических алгоритмов, включая как сами алгоритмы оптимизации, так и стратегии выбора начальных параметров. Более развитые алгоритмы подстраивают свою скорость обучения или используют информацию о вторых производных функции стоимости. И в заключение приведем обзор нескольких стратегий, заключающихся во включении простых алгоритмов оптимизации в высокоуровневые процедуры.

8.1. Чем обучение отличается от чистой оптимизации

Алгоритмы оптимизации, используемые для обучения глубоких моделей, отличаются от традиционных алгоритмов оптимизации в нескольких отношениях. Машинное обучение обычно работает не напрямую. В большинстве ситуаций нас интересует не-

которая мера качества P, которая определена относительно тестового набора и может оказаться вычислительно неприступной. Поэтому мы оптимизируем P косвенно. Мы уменьшаем другую функцию стоимости $J(\boldsymbol{\theta})$ в надежде, что при этом улучшится и P. Это резко отличается от чистой оптимизации, где минимизация J и есть конечная цель. Кроме того, алгоритмы оптимизации для обучения глубоких моделей обычно включают специализации для конкретной структуры целевых функций.

Типичную функцию стоимости можно представить в виде среднего по обучающему набору:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y), \tag{8.1}$$

где L – функция потерь на одном примере, $f(\boldsymbol{x};\boldsymbol{\theta})$ – предсказанный выход для входа $\boldsymbol{x}$, а $\hat{p}_{\text{data}}$ – эмпирическое распределение. В случае обучения с учителем y – ассоциированная с входом метка. В этой главе мы будем рассматривать нерегуляризированное обучение с учителем, когда аргументами L являются $f(\boldsymbol{x};\boldsymbol{\theta})$ и y. Этот случай тривиально обобщается, например, на включение в качестве аргументов $\boldsymbol{\theta}$ или $\boldsymbol{x}$ или на исключение y из числа аргументов с целью разработки различных видов регуляризации или обучения без учителя.

Уравнение (8.1) определяет целевую функцию относительно обучающего набора. Но мы обычно предпочитаем минимизировать соответствующую целевую функцию, в которой математическое ожидание берется по *порождающему данные распределению* p_{data}, а не просто по конечному обучающему набору:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y). \tag{8.2}$$

8.1.1. Минимизация эмпирического риска

Цель алгоритма машинного обучения – уменьшить математическое ожидание ошибки обобщения, описываемое формулой (8.2). Эта величина называется **риском**. Подчеркнем еще раз, что математическое ожидание берется по истинному распределению p_{data}. Если бы мы знали истинное распределение $p_{\text{data}}(\boldsymbol{x}, y)$, то минимизация риска была бы задачей оптимизации, решаемой с помощью алгоритма оптимизации. Но когда $p_{\text{data}}(\boldsymbol{x}, y)$ неизвестно, а есть только обучающий набор примеров, мы имеем задачу машинного обучения.

Простейший способ преобразовать задачу машинного обучения в задачу оптимизации – минимизировать ожидаемые потери на обучающем наборе. Это значит, что мы заменяем истинное распределение $p(\boldsymbol{x}, y)$ эмпирическим распределением $\hat{p}(\boldsymbol{x}, y)$, определяемым по обучающему набору. И теперь требуется минимизировать **эмпирический риск**:

$$\mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{\text{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}), \tag{8.3}$$

где m – количество обучающих примеров.

Процесс обучения, основанный на минимизации этой средней ошибки обучения, называется **минимизацией эмпирического риска**. В такой постановке машинное обучение все еще очень похоже на чистую оптимизацию. Вместо оптимизации риска напрямую мы оптимизируем эмпирический риск и надеемся, что и риск тоже заметно уменьшится. Существуют теоретические результаты, устанавливающие условия, при которых можно ожидать того или иного уменьшения истинного риска.

Тем не менее минимизация эмпирического риска уязвима для переобучения. Модели высокой емкости могут попросту запомнить обучающий набор. Во многих случаях минимизация эмпирического риска практически неосуществима. Самые эффективные современные алгоритмы оптимизации основаны на градиентном спуске, но для многих полезных функций потерь, например бинарной, производная малоинтересна (либо равна нулю, либо не определена). Из-за этих двух проблем минимизация эмпирического риска редко применяется в контексте глубокого обучения. Вместе нее используется несколько иной подход, при котором фактически оптимизируемая величина еще сильнее отличается от той, которую мы хотели бы оптимизировать на самом деле.

8.1.2. Суррогатные функции потерь и ранняя остановка

Иногда реально интересующая нас функция потерь (скажем, ошибка классификации) и та, что может быть эффективно оптимизирована, – «две большие разницы». Например, задача точной минимизации ожидаемой бинарной функции потерь обычно неразрешима (она экспоненциально зависит от размерности входных данных), даже для линейного классификатора (Marcotte and Savard, 1992). В таких случаях оптимизируют **суррогатную функцию потерь**, выступающую в роли заместителя истинной, но обладающую рядом преимуществ. Например, в качестве суррогата бинарной функции потерь часто берут отрицательное логарифмическое правдоподобие правильного класса. Отрицательное логарифмическое правдоподобие позволяет модели оценить условную вероятность классов при известном выходе, и если модель делает это хорошо, то она сможет выбрать классы, дающие наименьшую ожидаемую ошибку классификации.

В некоторых случаях суррогатная функция потерь позволяет достичь даже больших успехов в обучении. Например, на тестовом наборе бинарная потеря продолжает уменьшаться еще долго после того, как на обучающем наборе достигла нуля, если обучение производилось с использованием суррогата в виде логарифмического правдоподобия. Объясняется это тем, что даже когда ожидаемая бинарная потеря равна 0, робастность классификатора можно еще улучшить, отодвинув классы дальше друг от друга и получив тем самым более уверенный и надежный классификатор, который извлекает больше информации из обучающих данных, чем было бы возможно в случае простой минимизации средней бинарной потери на обучающем наборе.

Очень важное различие между оптимизацией вообще и применяемой в алгоритмах обучения заключается в том, что алгоритмы обучения обычно останавливаются не в локальном минимуме. Вместо этого алгоритм, как правило, минимизирует суррогатную функцию потерь, но останавливается, когда выполнено условие сходимости, основанное на идее ранней остановки (раздел 7.8). Типичное условие ранней остановки основано на истинной функции потерь, например вычислении бинарной функции потерь на контрольном наборе, и предназначено для того, чтобы остановить работу алгоритма, когда возникает угроза переобучения. Обучение зачастую заканчивается, когда производные суррогатной функции потерь все еще велики, и этим разительно отличается от чистой оптимизации, при которой считается, что алгоритм сошелся, если градиент стал очень малым.

8.1.3. Пакетные и мини-пакетные алгоритмы

Еще одно отличие алгоритмов машинного обучения от общих алгоритмов оптимизации состоит в том, что целевая функция обычно представлена в виде суммы по обучающим примерам. Типичный алгоритм оптимизации в машинном обучении вычисляет

каждое обновление параметров, исходя из ожидаемого значения функции стоимости, оцениваемого только по подмножеству членов полной функции стоимости.

Например, оценка максимального правдоподобия, рассматриваемая в логарифмическом пространстве, представлена в виде суммы по всем примерам:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \tag{8.4}$$

Максимизация этой суммы эквивалентна максимизации математического ожидания эмпирического распределения, определяемого обучающим набором:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.5}$$

Большинство свойств целевой функции J, используемой чуть ли не во всех наших алгоритмах оптимизации, также выражается в терминах математического ожидания по обучающему набору. Например, чаще всего используется ее градиент:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.6}$$

Вычисление точного значения этого математического ожидания обошлось бы очень дорого, потому что для этого нужно вычислить модель на каждом примере из набора данных. На практике можно случайно выбрать небольшое число примеров и усреднить только по ним.

Напомним, что стандартная ошибка среднего (формула 5.46), оцененная по выборке объема n, равна $\sigma/\sqrt{n}$, где σ – истинное стандартное отклонение выборки. Знаменатель $\sigma/\sqrt{n}$ показывает, что точность оценки градиента с увеличением объема выборки растет медленнее, чем линейно. Сравним две гипотетические оценки градиента, одна на основе 100 примеров, другая – 10 000. Для вычисления второй оценки потребуется в 100 раз больше времени, но стандартная ошибка среднего уменьшится только в 10 раз. Большинство алгоритмов оптимизации сходится гораздо быстрее (в терминах общего времени вычислений, а не числа обновлений), если им позволено быстро вычислять приближенные оценки градиента вместо медленного вычисления точного значения.

Еще одно сообщение в пользу статистического оценивания градиента по небольшой выборке связано с избыточностью обучающего набора. В худшем случае все m примеров в обучающем наборе в точности совпадают. Оценка градиента по выборке дала бы правильное значение, взяв всего один пример, т. е. было бы затрачено в m раз меньше времени, чем при наивном подходе. На практике нам вряд ли встретится худший случай, но все же можно найти много примеров, дающих очень похожий вклад в градиент.

Алгоритмы оптимизации, в которых используется весь обучающий пакет, называются **пакетными**, или **детерминированными**, градиентными методами, поскольку обрабатывают сразу все примеры одним большим пакетом. Эта терминология может вызывать путаницу, потому что слово «пакет» часто употребляется также для обозначения мини-пакета, применяемого в алгоритме стохастического градиентного спуска. Как правило, термин «пакетный градиентный спуск» подразумевает использование всего обучающего набора, а термин «пакет», применяемый для описания группы примеров, таких коннотаций ***уточнить слово*** не имеет. Например, словосочетание «размер пакета» часто означает размер мини-пакета.

Алгоритмы оптимизации, в которых используется по одному примеру за раз, иногда называют **стохастическими**, или **онлайновыми**, методами. Термин «онлайновый»

обычно резервируется для случая, когда примеры выбираются из непрерывного потока, а не из обучающего набора фиксированного размера, по которому можно совершать несколько проходов.

Большинство алгоритмов, используемых в глубоком обучении, находится где-то посередине – число примеров в них больше одного, но меньше размера обучающего набора. Традиционно они назывались **мини-пакетными,** или **мини-пакетными стохастическими**, методами, а сейчас – просто **стохастическими**.

Канонический пример стохастического метода – стохастический градиентный спуск, который подробно будет описан в разделе 8.3.1.

На размер мини-пакета оказывают влияние следующие факторы:

- чем больше пакет, тем точнее оценка градиента, но зависимость хуже линейной;
- если пакет очень мал, то не удается в полной мере задействовать преимущества многоядерной архитектуры. Поэтому существует некий абсолютный минимум размера пакета – такой, что обработка мини-пакетов меньшего размера не дает никакого выигрыша во времени;
- если все примеры из пакета нужно обрабатывать параллельно (так обычно и бывает), то размер пакета лимитирован объемом памяти. Для многих аппаратных конфигураций размер пакета – ограничивающий фактор;
- для некоторых видов оборудования оптимальное время выполнения достигается при определенных размерах массива. Так, для GPU наилучшие результаты получаются, когда размер пакета – степень 2. Типичный пакет имеет размер от 32 до 256, а для особо больших моделей иногда пробуют 16;
- небольшие пакеты могут дать эффект регуляризации (Wilson and Martinez, 2003), быть может, из-за шума, который они вносят в процесс обучения. Ошибка обобщения часто оказывается наилучшей для пакета размера 1. Но для обучения с таким маленьким размером пакета нужна небольшая скорость обучения для обеспечения устойчивости из-за высокой дисперсии оценки градиента. Общее время работы может оказаться очень большим из-за увеличения числа шагов – как из-за пониженной скорости обучения, так и потому, что для перебора всего обучающего набора требуется больше шагов.

В зависимости от вида алгоритма используется разная информация из мини-пакета, причем разными способами. Одни алгоритмы более чувствительны к ошибке выборки, чем другие, либо потому что в них используется информация, которую трудно оценить точно на небольшой выборке, либо потому что информация используется так, что ошибка выборки усиливается. Методы, которые вычисляют обновления только на основе градиента $\boldsymbol{g}$, обычно сравнительно устойчивы и могут работать с пакетами небольшого размера, порядка 100. Методы второго порядка, в которых используется также матрица Гессе $\boldsymbol{H}$ и которые вычисляют такие обновления, как $\boldsymbol{H}^{-1}\boldsymbol{g}$, обычно нуждаются в пакетах гораздо большего размера, порядка 10 000. Такие большие пакеты нужны, чтобы свести к минимуму флуктуации в оценках $\boldsymbol{H}^{-1}\boldsymbol{g}$. Предположим, что $\boldsymbol{H}$ оценена идеально, но ее число обусловленности плохое. Тогда умножение на $\boldsymbol{H}$ или на обратную к ней матрицу усиливает уже имеющиеся ошибки, в данном случае ошибки оценки $\boldsymbol{g}$. Следовательно, очень малые изменения в оценке $\boldsymbol{g}$ могут привести к большим изменениям при обновлении $\boldsymbol{H}^{-1}\boldsymbol{g}$, хотя оценка $\boldsymbol{H}$ точна. На самом деле оценка $\boldsymbol{H}$ – всего лишь аппроксимация, поэтому ошибка обновления $\boldsymbol{H}^{-1}\boldsymbol{g}$ будет даже больше, чем вследствие одного лишь применения плохо обусловленной операции к оценке $\boldsymbol{g}$.

Важно также, чтобы мини-пакеты выбирались случайно. Для вычисления несмещенной оценки ожидаемого градиента по выборке необходимо, чтобы примеры были независимы. Мы также хотим, чтобы две последовательные оценки градиента были независимы друг от друга, поэтому два последовательных мини-пакета примеров тоже должны быть независимы. Многие наборы данных естественно упорядочены так, что между последовательными примерами имеется высокая корреляция. Например, длинный список результатов анализа крови, скорее всего, организован так, что сначала идут пять анализов одного пациента, взятых в разные моменты времени, затем – три анализа второго пациента и т. д. Если бы мы выбирали примеры из такого набора, то каждый мини-пакет оказался бы очень сильно смещенным, т. к. представлял бы преимущественно одного пациента из многих присутствующих в наборе данных. В тех случаях, когда порядок примеров в наборе не случаен, необходимо перетасовать пакет, прежде чем формировать мини-пакеты. Для очень больших наборов, насчитывающих миллиарды примеров, выбирать примеры по-настоящему случайно при каждом построении мини-пакета не всегда возможно. К счастью, на практике обычно достаточно перетасовать набор один раз и затем хранить его в таком виде. При этом получается фиксированный набор возможных мини-пакетов последовательных примеров, которым вынуждены будут пользоваться все обучаемые впоследствии модели, и каждая модель будет видеть примеры в одном и том же порядке при проходе по обучающим данных. Но похоже, что такое отклонение от истинно случайного выбора не оказывает значимого негативного эффекта. Тогда как полное пренебрежение перетасовкой примеров способно серьезно снизить эффективность алгоритма.

Во многих задачах оптимизации в машинном обучении примеры структурированы достаточно хорошо, чтобы можно было параллельно вычислять несколько обновлений по разным примерам. Иными словами, мы можем вычислять обновление, минимизирующее $J(\boldsymbol{X})$ для одного мини-пакета $\boldsymbol{X}$, одновременно с вычислением обновления для нескольких других мини-пакетов. Такие асинхронные параллельные распределенные подходы обсуждаются подробнее в разделе 12.1.3.

Интересным обоснованием мини-пакетного стохастического градиентного спуска является тот факт, что он происходит в направлении градиента истинной *ошибки обобщения* (уравнение 8.2), при условии что примеры не повторяются. В большинстве реализаций этого алгоритма набор данных перетасовывается один раз, после чего по нему производится несколько проходов. На первом проходе каждый мини-пакет используется для вычисления несмещенной оценки истинной ошибки обобщения. На втором проходе оценка становится смещенной, потому что получена повторной выборкой уже использованных значений, а не новых примеров из порождающего распределения.

Тот факт, что алгоритм стохастического градиентного списка действительно минимизирует ошибку обобщения, отчетливее всего виден в онлайновом обучении, когда примеры или мини-пакеты выбираются из потока данных. Иными словами, обучаемая модель не получает обучающего набора фиксированного размера, а, подобно живому существу, в каждый момент времени видит новый пример; при этом каждый пример $(\boldsymbol{x}, y)$ поступает из порождающего распределения $p_{\text{data}}(\boldsymbol{x}, y)$. В такой ситуации примеры никогда не повторяются, каждое испытание – честная выборка из p_{data}.

Эквивалентность проще всего установить, когда $\boldsymbol{x}$ и y – дискретные величины. В таком случае ошибку обобщения (8.2) можно переписать в виде суммы:

$$J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) L(f(\boldsymbol{x}; \boldsymbol{\theta}), y), \tag{8.7}$$

и точный градиент равен

$$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J^{*}(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}; \boldsymbol{\theta}), y). \tag{8.8}$$

Мы уже демонстрировали тот же факт для логарифмического правдоподобия в уравнениях (8.5) и (8.6); теперь мы видим, что это справедливо и для других функций L. Аналогичный результат можно доказать для случая, когда $\boldsymbol{x}$ и y непрерывны, если наложить на p_{data} и L не слишком обременительные ограничения.

Таким образом, мы можем получить несмещенную оценку точного градиента ошибки обобщения, выбрав мини-пакет примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ с соответственными метками $y^{(i)}$ из порождающего распределения p_{data}, а затем вычислив для этого мини-пакета градиент функции потерь относительно параметров:

$$\hat{\boldsymbol{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \tag{8.9}$$

Для обновления $\boldsymbol{\theta}$ в направлении $\hat{\boldsymbol{g}}$ выполняется СГС по ошибке обобщения.

Разумеется, эта интерпретация справедлива, только когда примеры не используются повторно. Тем не менее обычно для получения наилучших результатов стоит сделать несколько проходов по обучающему набору, если только он не слишком велик. Если используется несколько таких периодов, то лишь в первом спуск происходит в направлении несмещенного градиента ошибки обобщения, но, конечно, дополнительные периоды обычно дают достаточный выигрыш в плане уменьшения ошибки обучения, чтобы перевесить вред от увеличения разрыва между ошибкой обучения и ошибкой тестирования.

В тех случаях, когда размер набора данных растет быстрее, чем вычислительные ресурсы для его обработки, все чаще в машинном обучении переходят к практике, когда каждый обучающий пример используется ровно один раз, или даже производится неполный проход по обучающему набору. Если обучающий набор очень велик, то переобучение перестает быть проблемой, и на первый план выходят проблемы недообучения и вычислительной эффективности. См. также работу Bottou and Bousquet (2008), где обсуждается влияние вычислительных узких мест на ошибку обобщения при увеличении числа обучающих примеров.

8.2. Проблемы оптимизации нейронных сетей

В общем случае оптимизация – чрезвычайно трудная задача. Традиционно в машинном обучении избегали сложностей общей оптимизации за счет тщательного выбора целевой функции и ограничений, гарантирующих выпуклость задачи оптимизации. При обучении нейронных сетей приходится сталкиваться с общим невыпуклым случаем. Но даже выпуклая оптимизация не обходится без осложнений. В этом разделе мы дадим обзор нескольких наиболее заметных проблем оптимизации, возникающих в процессе обучения глубоких моделей.

8.2.1. Плохая обусловленность

Ряд проблем возникает даже при оптимизации выпуклых функций. Самая известная из них – плохая обусловленность матрицы Гессе $\boldsymbol{H}$. Это очень общая проблема, присущая большинству методов численной оптимизации, все равно, выпуклой или нет, она подробно описана в разделе 4.3.1.

Считается, что проблема плохой обусловленности присутствует во всех задачах обучения нейронных сетей. Она может проявляться в «застревании» СГС в том смысле, что даже очень малые шаги увеличивают функцию стоимости.

Напомним, что согласно формуле (4.9) разложение функции стоимости в ряд Тейлора до членов второго порядка показывает, что шаг градиентного спуска величиной $-\varepsilon \boldsymbol{g}$ увеличивает стоимость на

$$^1/_2\varepsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g} - \varepsilon \boldsymbol{g}^\top \boldsymbol{g}. \tag{8.10}$$

Плохая обусловленность градиента становится проблемой, когда $^1/_2\varepsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ больше $\varepsilon \boldsymbol{g}^\top \boldsymbol{g}$. Чтобы понять, страдает ли задача обучения нейронной сети от плохой обусловленности, можно понаблюдать за квадратом нормы градиента $\boldsymbol{g}^\top \boldsymbol{g}$ и членом $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$. Во многих случаях норма градиента не сильно уменьшается за время обучения, тогда как член $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ возрастает больше, чем на порядок. В результате обучение происходит очень медленно, несмотря на большой градиент, т. к. приходится уменьшать скорость обучения, чтобы компенсировать еще большую кривизну. На рис. 8.1 приведен пример, когда градиент значительно увеличивается в ходе успешного обучения нейронной сети.

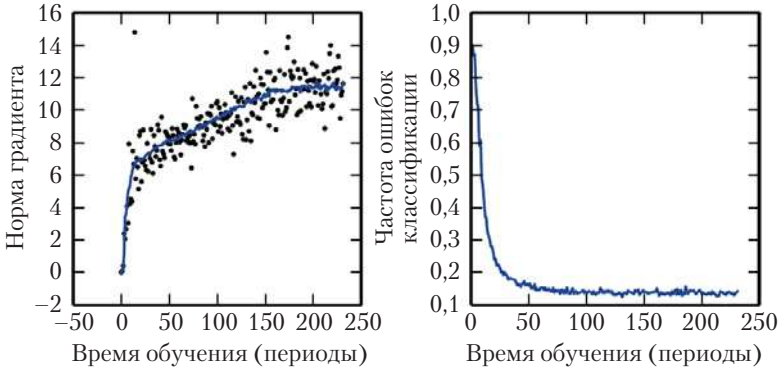

Рис. 8.1 ❖ Градиентный спуск часто не находит никакой критической точки. В этом примере норма градиента возрастает на протяжении всего процесса обучения сверточной нейронной сети для обнаружения объектов. (*Слева*) На диаграмме рассеяния показана зависимость вычисленной нормы градиента от времени. Для наглядности показана лишь одна норма на каждый период. Скользящее среднее всех норм градиента показано сплошной линией. Очевидно, что норма градиента со временем возрастает, а не убывает, как было бы, если бы процесс обучения сходился к критической точке. (*Справа*) Несмотря на возрастание градиента, процесс обучения можно считать успешным. Ошибка классификации на контрольном наборе убывает до низкого уровня

Хотя плохая обусловленность характерна не только для обучения нейронных сетей, некоторые методы борьбы с ней, используемые в других контекстах, к нейронным сетям плохо применимы. Например, метод Ньютона – отличное средство минимизации выпуклых функций с плохо обусловленными гессианами, но, как мы покажем в следующих разделах, для применения к нейронным сетям этот метод нуждается в существенной модификации.

8.2.2. Локальные минимумы

Одна из самых важных черт выпуклой оптимизации состоит в том, что такую задачу можно свести к задаче нахождения локального минимума. Гарантируется, что любой локальный минимум одновременно является глобальным. У некоторых выпуклых функций в нижней части графика имеется не единственный глобальный минимум, а целый плоский участок. Однако любая точка на плоском участке является допустимым решением. При оптимизации выпуклой функции мы точно знаем, что, обнаружив критическую точку любого вида, мы нашли хорошее решение.

У невыпуклых функций, в частности нейронных сетей, локальных минимумов может быть несколько. Более того, почти у любой глубокой модели гарантированно имеется множество локальных минимумов. Впрочем, как мы увидим, это не всегда является серьезной проблемой.

Нейронные сети и вообще любые модели с несколькими эквивалентно параметризованными латентными переменными имеют несколько локальных минимумов из-за проблемы **идентифицируемости модели**. Говорят, что модель идентифицируемая, если существует достаточно большой обучающий набор, который может исключить все конфигурации параметров модели, кроме одной. Модели с латентными переменными часто не являются идентифицируемыми, потому что мы можем получить эквивалентные модели, меняя латентные переменные местами. Например, можно было бы взять нейронную сеть и модифицировать слой 1, заменив входящий вектор весов для блока i входящим векторов весов для блока j и наоборот, а затем проделав то же самое для исходящих векторов весов. Если имеется m слоев по n блоков в каждом, то существует $n!^m$ способов упорядочить скрытые блоки. Такой вид неидентифицируемости называется **симметрией пространства весов**.

Помимо симметрии пространства весов, во многих разновидностях нейронных сетей есть и другие причины неидентифицируемости. Например, в любой сети с блоками линейной ректификации или maxout-блоками можно умножить все входящие веса и смещения блока на α, одновременно умножив исходящие веса на $1/\alpha$. Это означает, что если функция стоимости не включает таких членов, как снижение весов, которые напрямую зависят от весов, а не от выходов модели, то все локальные минимумы сети лежат на $(m \times n)$-мерном гиперболоиде эквивалентных локальных минимумов.

Проблема идентифицируемости модели означает, что функция стоимости нейронной сети может иметь очень большое, даже несчетное, множество локальных минимумов. Однако все локальные минимумы, проистекающие из неидентифицируемости, эквивалентны между собой с точки зрения значения функции стоимости. Поэтому такое проявление невыпуклости не составляет проблемы.

Локальные минимумы становятся проблемой, если значение функции стоимости в них велико, по сравнению со значением в глобальном минимуме. Можно построить небольшую нейронную сеть, даже без скрытых блоков, в которой стоимость в локальных минимумах будет выше, чем в глобальном (Sontag and Sussman, 1989; Brady et al., 1989; Gori and Tesi, 1992). Если локальные минимумы с высокой стоимостью встречаются часто, то градиентные алгоритмы оптимизации сталкиваются с серьезной проблемой.

Вопрос о том, много ли локальных минимумов с высокой стоимостью в практически интересных сетях и наталкиваются ли на них алгоритмы оптимизации, остается открытым. В течение многих лет среди практиков бытовало мнение, что локальные минимумы – распространенная проблема, преследующая оптимизацию нейронных

сетей. Но сегодня так не кажется. В этой области ведутся активные исследования, но специалисты склоняются к мнению, что для достаточно больших нейронных сетей в большинстве локальных минимумов значение функции стоимости мало и что важно не столько найти глобальный минимум, сколько какую-нибудь точку в пространстве параметров, в которой стоимость низкая, пусть и не минимальная (Saxe et al., 2013; Dauphin et al., 2014; Goodfellow et al., 2015; Choromanska et al., 2014).

Многие специалисты-практики приписывают почти все трудности, связанные с оптимизацией нейронных сетей, локальным минимумам. Мы призываем их тщательнее изучать конкретные задачи. Чтобы исключить локальные минимумы как возможную причину проблем, имеет смысл построить график зависимости нормы градиента от времени. Если норма градиента не убывает почти до нуля, то проблема не в локальных минимумах и вообще не в критических точках. В пространствах высокой размерности установить с полной определенностью, что корень зла – локальные минимумы, бывает очень трудно. Малые градиенты характерны для многих особенностей строения, помимо локальных минимумов.

8.2.3. Плато, седловые точки и другие плоские участки

Для многих невыпуклых функций в многомерных пространствах локальные минимумы (и максимумы) встречаются гораздо реже других точек с нулевым градиентом: седловых точек. В одних точках в окрестности седловой стоимость выше, чем в седловой точке, в других – ниже. В седловой точке матрица Гессе имеет как положительные, так и отрицательные собственные значения. В точках, лежащих вдоль собственных векторов с положительными собственными значениями, стоимость выше, чем в седловой точке, а в точках, лежащих вдоль собственных векторов с отрицательными собственными значениями, – ниже. Можно считать, что седловая точка является локальным минимумом в одном сечении графика функции стоимости и локальным максимумом – в другом. Это иллюстрирует рис. 4.5.

Многие классы случайных функций демонстрируют следующее поведение: в пространствах низкой размерности локальные минимумы встречаются часто, а в пространствах большей размерности они редкость, зато часто встречаются седловые точки. Для функции $f : \mathbb{R}^n \rightarrow \mathbb{R}$ такого типа ожидаемое отношение числа седловых точек к числу локальных максимумов растет экспоненциально с ростом n. Чтобы интуитивно понять причины такого поведения, заметим, что в локальном минимуме все собственные значения матрицы Гессе положительны. В седловой точке у матрицы Гессе есть как положительные, так и отрицательные собственные значения. Представьте себе, что знак собственного значения определяется подбрасыванием монеты. В одномерном случае для получения локального минимума достаточно, чтобы один раз выпал орел. А в n-мерном случае вероятность, что n раз подряд выпадет орел, экспоненциально убывает. Обзор теоретических работ на эту тему см. в Dauphin et al. (2014).

У многих случайных функций есть удивительное свойство: вероятность положительности собственных значений матрицы Гессе возрастает при приближении к областям низкой стоимости. В нашей аналогии с подбрасыванием монеты это означает, что вероятность n раз подряд выкинуть орла выше, если мы находимся в критической точке с низкой стоимостью. Это также означает, что локальные минимумы с низкой стоимостью гораздо вероятнее, чем с высокой. Критические точки с высокой стоимостью с куда большей вероятностью являются седловыми точками. А критические точки с очень высокой стоимостью, скорее всего, являются локальными максимумами.

Это верно для многих классов случайных функций. А для нейронных сетей? В работе Baldi and Hornik (1989) теоретически доказано, что мелкие автокодировщики (описанные в главе 14 сети прямого распространения, обученные копировать вход в выход) без нелинейностей имеют глобальные минимумы и седловые точки, но не имеют локальных минимумов со стоимостью выше, чем в глобальном минимуме. Не приводя доказательства, они заметили, что эти результаты обобщаются и на более глубокие сети без нелинейностей. Выходом такой сети является линейная функция от входа, но они полезны в качестве модели нелинейных нейронных сетей, поскольку функция потерь такой сети – невыпуклая функция своих параметров. Подобные сети, по существу, представляют собой просто композицию нескольких матриц. В работе Saxe et al. (2013) приведены точные решения для полной динамики обучения такой сети и показано, что обучение таких моделей улавливает многие качественные особенности, наблюдаемые при обучении глубоких моделей с нелинейными функциями активации. В работе Dauphin et al. (2014) экспериментально показано, что у реальных нейронных сетей также имеются функции потерь, содержащие очень много седловых точек с высокой стоимостью. В работе Choromanska et al. (2014) приведены дополнительные теоретические аргументы, доказывающие, что это справедливо еще для одного класса многомерных случайных функций, родственного нейронным сетям.

Каковы последствия изобилия седловых точек для алгоритмов обучения? В случае оптимизации первого порядка, когда используется только информация о градиенте, ситуация неясна. Градиент часто оказывается очень мал в окрестности седловой точки. С другой стороны, есть эмпирические свидетельства в пользу того, что метод градиентного спуска во многих случаях способен выйти из седловой точки. В работе Goodfellow et al. (2015) наглядно показано несколько траекторий обучения современных нейронных сетей, один из таких примеров приведен на рис. 8.2. На этих рисунках видно уплощение функции стоимости вблизи выраженной седловой точки, где все веса равны нулю, но видно и то, что траектория градиентного спуска быстро покидает этот участок. В той же работе высказывается предположение, что можно аналитически доказать, что седловая точка отталкивает, а не притягивает траекторию непрерывного по времени градиентного спуска, но что ситуация может оказаться иной в более реалистичных случаях применения метода градиентного спуска.

Для метода Ньютона седловые точки представляют очевидную проблему. Идея алгоритма градиентного спуска – «спуск с горы», а не явный поиск критических точек. С другой стороны, метод Ньютона специально предназначен для поиска точек с нулевым градиентом. Без надлежащей модификации он вполне может найти седловую точку. Изобилие седловых точек в многомерных пространствах объясняет, почему методы второго порядка не смогли заменить градиентный спуск в обучении нейронных сетей. В работе Dauphin et al. (2014) описан **бесседловой метод Ньютона** (saddle-free Newton method) для оптимизации второго порядка и показано, что он значительно улучшает традиционный вариант. Методы второго порядка все еще с трудом масштабируются на большие нейронные сети, но если этот бесседловой метод удастся масштабировать, то он сулит интересные перспективы.

Помимо минимумов и седловых точек, существуют и другие виды точек с нулевым градиентом. С точки зрения оптимизации, максимумы очень похожи на седловые точки – многие алгоритмы не притягиваются к ним, но немодифицированный метод Ньютона не из их числа. Для многих классов случайных функций в многомерном пространстве максимумы – такая же экспоненциальная редкость, как и минимумы.

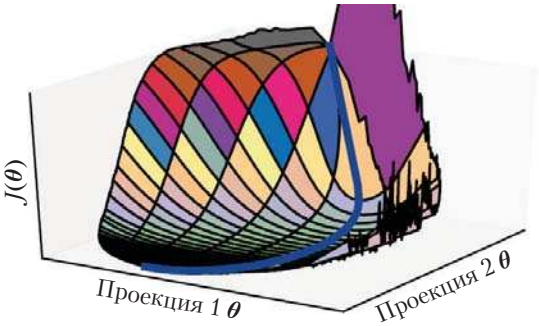

Рис. 8.2 ❖ Визуализация функции стоимости нейронной сети. Похожие визуализации характерны для нейронных сетей прямого распространения, а также сверточных и рекуррентных, применяемых в реальных задачах распознавания объектов и обработки естественных языков. Как ни странно, на этих визуализациях обычно не встретишь много бросающихся в глаза препятствий. До триумфа алгоритма стохастического градиентного спуска в применении к обучению очень больших моделей, датируемого примерно 2012 годом, считалось, что поверхности функций стоимости нейронных сетей обладают куда более невыпуклой структурой, чем видно на этих проекциях. Основное присутствующее здесь препятствие – седловая точка высокой стоимости вблизи начальных значений параметров, но, как показывает синяя линия, траектория обучения СГС быстро покидает эту седловую точку. Основное время затрачено на пересечение сравнительно плоской долины функции стоимости, наверное, вследствие высокого шума при вычислении градиента, плохой обусловленности гессиана в этой области или просто из-за необходимости обойти высокую «гору», видную на рисунке, по огибающей дуге. Изображение взято из работы Goodfellow et al. (2015) с разрешения авторов

Могут также существовать широкие плоские области с постоянным значением. В этих областях равны нулю и градиент, и гессиан. Такие вырожденные участки – серьезная проблема для всех алгоритмов численной оптимизации. В выпуклой задаче широкая плоская область должна целиком состоять из глобальных минимумов, но в общем случае ей могут соответствовать и большие значения целевой функции.

8.2.4. Утесы и резко растущие градиенты

В нейронных сетях с большим числом слоев часто встречаются очень крутые участки, напоминающие утесы (рис. 8.3). Это связано с перемножением нескольких больших весов. На стене особенно крутого утеса шаг обновления градиента может привести к очень сильному изменению параметров, что обычно заканчивается «срывом» с утеса.

Утес представляет опасность вне зависимости от того, приближаемся мы к нему сверху или снизу, но, по счастью, самых печальных последствий можно избежать с помощью эвристической техники **отсечения градиента**, описанной в разделе 10.11.1. Основная идея – вспомнить, что градиент определяет не оптимальный размер шага, а лишь оптимальное направление в бесконечно малой области. Когда традиционный алгоритм градиентного спуска предлагает сделать очень большой шаг, вмешивается эвристика отсечения, и в результате шаг уменьшается, так что становится менее

вероятным выход за пределы области, в которой градиент указывает приближенное направление самого крутого спуска. Утесы чаще всего встречаются в функциях стоимости рекуррентных нейронных сетей, поскольку в таких моделях вычисляется произведение большого числа множителей, по одному на каждый временной шаг. Поэтому чем длиннее временная последовательность, тем больше количество перемножений.

8.2.5. Долгосрочные зависимости

Еще одна трудность для алгоритмов оптимизации нейронной сети возникает, когда граф вычислений становится очень глубоким. Такие графы характерны для многослойных сетей прямого распространения, а также для рекуррентных сетей (глава 10), в которых очень глубокий граф вычислений создается в результате применения одной и той же операции на каждом шаге длинной временной последовательности. Повторное применение одних и тех же параметров вызывает особенно трудно преодолимые сложности.

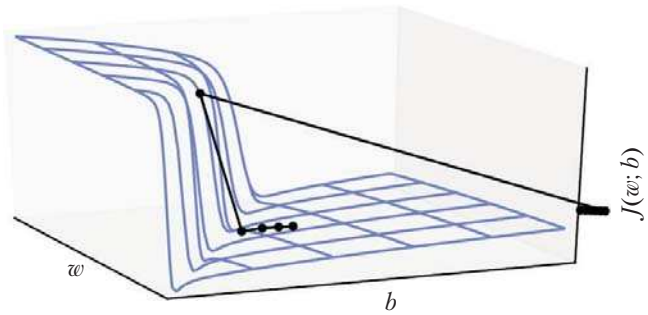

Рис. 8.3 ❖ Целевая функция сильно нелинейной глубокой нейронной сети или рекуррентной сети часто характеризуется резкими нелинейностями в пространстве параметров, возникающими из-за перемножения нескольких параметров. В местах таких нелинейностей значения производных очень велики. Когда параметры приближаются к подобному утесу, шаг обновления в методе градиентного спуска может сдвинуть параметры очень далеко, при этом может потеряться все, чего удалось достичь в ходе предшествующей оптимизации. Рисунок взят из работы Pascanu et al. (2013) с разрешения авторов

Например, предположим, что граф вычислений содержит путь, состоящий из повторных умножений на матрицу $\boldsymbol{W}$. Выполнение t шагов эквивалентно умножению на $\boldsymbol{W}^t$. Пусть спектральное разложение $\boldsymbol{W}$ имеет вид $\boldsymbol{W} = \boldsymbol{V}\,\mathrm{diag}(\boldsymbol{\lambda})\,\boldsymbol{V}^{-1}$. В этом случае легко видеть, что

$$\boldsymbol{W}^t = (\boldsymbol{V}\,\mathrm{diag}(\boldsymbol{\lambda})\,\boldsymbol{V}^{-1})^t = \boldsymbol{V}\,\mathrm{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}. \tag{8.11}$$

Все собственные значения λ_i, кроме близких к 1 по абсолютной величине, либо резко возрастают (если их абсолютная величина больше 1), либо почти обращаются в 0 (если абсолютная величина меньше 1). Когда говорят о **проблеме исчезающего и взрывного градиента**, имеют в виду, что в результате вычислений с таким графом градиенты также умножаются на коэффициент $\mathrm{diag}(\boldsymbol{\lambda})^t$. Если градиент обращается

в 0, то трудно понять, в каком направлении изменять параметры, чтобы улучшить функцию стоимости, а если резко возрастает, то обучение становится численно неустойчивым. Описанные выше утесистые структуры, для борьбы с которыми применяется отсечение градиента, – пример феномена взрывного градиента.

Повторное умножение на $\boldsymbol{W}$ на каждом временном шаге очень похоже на **степенной метод** нахождения наибольшего собственного значения матрицы $\boldsymbol{W}$ и соответствующего ему собственного вектора. С этой точки зрения неудивительно, что операция $\boldsymbol{x}^\top \boldsymbol{W}^t$ в конечном итоге отбрасывает все компоненты $\boldsymbol{x}$, ортогональные главному собственному вектору $\boldsymbol{W}$.

В рекуррентных сетях на каждом временном шаге используется одна и та же матрица $\boldsymbol{W}$, но в сетях прямого распространения это не так, поэтому даже в очень глубоких сетях прямого распространения, как правило, удается избежать проблемы исчезающего и взрывного градиента (Sussillo, 2014).

Мы отложим дальнейшее обсуждение проблем обучения рекуррентных сетей до раздела 10.7, когда опишем такие сети более детально.

8.2.6. Неточные градиенты

Большинство алгоритмов оптимизации исходит из предположения, что имеется доступ к точному градиенту или гессиану. На практике же обычно налицо только зашумленная или даже смещенная оценка этих величин. Почти все алгоритмы глубокого обучения опираются на выборочные оценки, по крайней мере в том, что касается использования мини-пакета обучающих примеров для вычисления градиента.

Бывает и так, что целевая функция, которую мы хотим минимизировать, вычислительно неразрешима. В таком случае неразрешимой обычно является и задача вычисления градиента, и тогда нам остается только аппроксимировать градиент. Такие проблемы чаще всего возникают в более сложных моделях, рассматриваемых в части III. Например, алгоритм сопоставительного расхождения (contrastive divergence) предлагает метод аппроксимации градиента функции логарифмического правдоподобия машины Больцмана.

Для компенсации неточной оценки градиента разработаны различные алгоритмы оптимизации нейронных сетей. Проблему можно обойти также путем выбора суррогатной функции потерь, которую проще аппроксимировать, чем истинную.

8.2.7. Плохое соответствие между локальной и глобальной структурами

Многие рассмотренные выше проблемы касаются свойств функции потерь в одной точке – трудно сделать следующий шаг, если $J(\boldsymbol{\theta})$ плохо обусловлена в текущей точке $\boldsymbol{\theta}$, или если $\boldsymbol{\theta}$ находится на стене утеса, или если $\boldsymbol{\theta}$ является седловой точкой, маскирующей возможность добиться улучшения путем «спуска с горы».

Все эти проблемы можно преодолеть в одной точке и тем не менее остаться «на бобах», если найденное направление наибольшего локального улучшения не ведет в сторону отдаленных областей с гораздо меньшей стоимостью.

В работе Goodfellow et al. (2015) утверждается, что длительность обучения определяется в первую очередь длиной траектории, ведущей к решению. На рис. 8.2 видно, что траектория обучения резко удлиняется из-за необходимости обогнуть по широкой дуге скалообразную структуру.

В центре многих исследований, посвященных трудностям оптимизации, находится вопрос о том, достигает ли обучение глобального минимума, локального минимума или седловой точки, но на практике нейронные сети не находят никакую критическую точку. На рис. 8.1 показано, что нейронная сеть часто не достигает области малых градиентов. Да, собственно, таких критических точек может и не оказаться. Например, у функции потерь $-\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$ может не быть точки глобального минимума, вместо этого она асимптотически приближается к некоторому значению, по мере того как модель становится более уверенной. Для классификатора с дискретными метками y и softmax-функцией $p(y \mid \boldsymbol{x})$ отрицательное логарифмическое правдоподобие может оказаться сколь угодно близким к нулю, если модель правильно классифицирует каждый обучающий пример, но никогда не принимает значения 0. Аналогично у вещественной модели $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ отрицательное логарифмическое правдоподобие может асимптотически приближаться к минус бесконечности – если $f(\boldsymbol{\theta})$ правильно предсказывает значение y для всех обучающих примеров, то алгоритм обучения будет неограниченно увеличивать β. На рис. 8.4 приведен пример, когда локальная оптимизация терпит неудачу при поиске хорошей функции стоимости даже в отсутствие локальных минимумов или седловых точек.

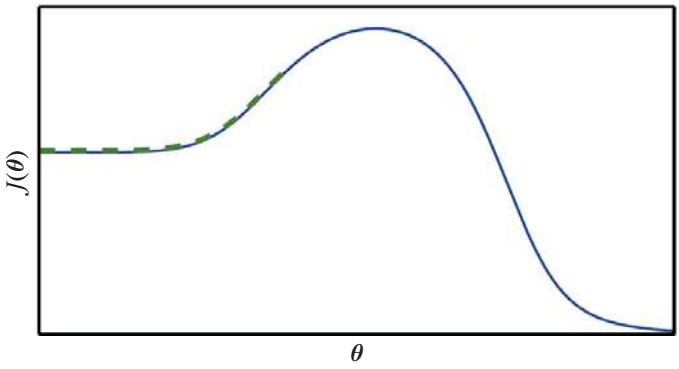

Рис. 8.4 ❖ Оптимизация, основанная на локальном спуске, может потерпеть неудачу, если локальное направление не ведет к глобальному решению. Здесь показано, как такое может произойти даже в отсутствие локальных минимумов или седловых точек. Функция стоимости в этом примере только асимптотически приближается к низким значениям, но не имеет минимумов. Проблема вызвана тем, что начальные значения выбраны не по ту сторону «горы», и алгоритм не может перебраться через нее. В многомерных пространствах алгоритмы обучения обычно способны обогнуть такие горы, но траектория может оказаться длинной, и обучение займет слишком много времени (см. рис. 8.2)

Будущим исследователям необходимо глубже разобраться в природе факторов, влияющих на длину траектории обучения, и лучше охарактеризовать результат процесса.

Многие из современных направлений исследований нацелены на поиск хороших начальных значений параметров в задачах с трудной глобальной структурой, а не на разработку алгоритмов с нелокальным перемещением.

Градиентный спуск и практически все алгоритмы, доказавшие эффективность при обучении нейронных сетей, основаны на небольших локальных шагах. В предыдущих разделах мы в основном говорили о трудностях вычисления правильного направления этих шагов. Не исключено, что некоторые свойства целевой функции, в т. ч. ее градиент, можно вычислить только приближенно, и оценка правильного направления будет либо смещенной, либо имеющей большую дисперсию. В таких случаях локальный спуск может дать или не дать разумно короткий путь к решению, но мы не можем последовать по этому пути. У целевой функции могут быть различные проблемы, например плохая обусловленность или разрывные градиенты, из-за которых область, в которой градиент дает хорошую модель целевой функции, очень мала. Тогда локальный спуск с шагами размера ε, возможно, и определяет разумно короткий путь к решению, но мы в состоянии вычислить направление локального спуска только с шагами размера $\delta \ll \varepsilon$. В этом случае может оказаться, что путь к решению, определяемый методом локального спуска, содержит слишком много шагов, и пройти по нему невозможно из-за высокой вычислительной стоимости. Иногда локальная информация не дает вообще никаких указаний, например когда у функции имеется широкий плоский участок или если нас угораздило попасть точно в критическую точку (такое обычно бывает, только если алгоритм находит критические точки в явном виде, как, например, метод Ньютона). В таких случаях локальный спуск вообще не определяет путь к решению. Бывает и так, что локальное перемещение оказывается слишком «жадным» и уводит нас по пути, который идет хоть и вниз, но в сторону от решения, как на рис. 8.4, или к решению, но по неоправданно длинной траектории, как на рис. 8.2. В настоящее время мы не понимаем, какие из этих проблем в наибольшей степени связаны с трудностями оптимизации нейронных сетей, и в этой области ведутся активные исследования.

Но вне зависимости от того, какие проблемы наиболее значимы, всех их можно избежать, если существует область пространства, связанная с решением относительно прямым путем, который может найти метод локального спуска, и если мы сумеем инициализировать параметры, так чтобы они попали в эту «хорошую» область. И эта точка зрения побуждает к исследованиям в области поиска хороших начальных значений для традиционных алгоритмов оптимизации.

8.2.8. Теоретические пределы оптимизации

Есть несколько теоретических результатов, показывающих, что существуют пределы качества у любого мыслимого алгоритма оптимизации для нейронных сетей (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997). Как правило, такие результаты бесполезны для практического использования нейронных сетей.

Ряд теоретических результатов относится только к случаю, когда блоки нейронной сети выводят дискретные значения. В большинстве нейронных сетей блоки выводят гладко возрастающие значения, благодаря чему и возможна оптимизация методом локального поиска. Другие результаты показывают, что существуют классы неразрешимых задач, но трудно сказать, попадает ли в такой класс данная конкретная задача. Есть также результаты, доказывающие невозможность найти решение для сети заданного размера, но на практике легко можно найти решение, используя сеть большего размера, для которой гораздо больше конфигураций параметров соответствуют приемлемому решению. Кроме того, в контексте обучения нейронных сетей нас обычно не интересует нахождение точного минимума функции, нужно лишь найти значение, достаточно малое для получения хорошей ошибки обобщения. Теоретически про-

анализировать, может ли алгоритм оптимизации достичь этой цели, исключительно трудно. Поэтому разработка более реалистичных границ качества алгоритмов оптимизации остается важной целью исследований по машинному обучению.

8.3. Основные алгоритмы

Мы уже познакомились с алгоритмом градиентного спуска (раздел 4.3), идея которого – перемещаться в направлении убывания градиента всего обучающего набора. Работу можно значительно ускорить, воспользовавшись стохастическим градиентным спуском для случайно выбранных мини-пакетов, как описано в разделах 5.9 и 8.1.3.

8.3.1. Стохастический градиентный спуск

Метод стохастического градиентного спуска (СГС) и его варианты – пожалуй, самые употребительные алгоритмы машинного обучения вообще и глубокого обучения в частности. Как было сказано в разделе 8.1.3, можно получить несмещенную оценку градиента, усреднив его по мини-пакету m независимых и одинаково распределенных примеров, выбранных из порождающего распределения.

В алгоритме 8.1 показано, как осуществить спуск вниз, пользуясь этой оценкой.

Алгоритм 8.1. Обновление на k-ой итерации стохастического градиентного спуска (СГС)

Require: скорость обучения ε_k
Require: Начальные значения параметров $\boldsymbol{\theta}$
  while критерий остановки не выполнен **do**
    Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
    Вычислить оценку градиента: $\hat{\boldsymbol{g}} \leftarrow +(1/m)\nabla_{\boldsymbol{\theta}}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon\hat{\boldsymbol{g}}$.
  end while

Основной параметр алгоритма СГС – скорость обучения. Ранее при описании СГС мы считали скорость обучения ε фиксированной. На практике же необходимо постепенно уменьшать ее со временем, поэтому будем обозначать ε_k скорость обучения на k-ой итерации.

Связано это с тем, что СГС-оценка градиента вносит источник шума (случайная выборка m обучающих примеров), который не исчезает, даже когда мы нашли минимум. Напротив, при использовании пакетного градиентного спуска истинный градиент полной функции стоимости уменьшается по мере приближения к минимуму и обращается в 0 в самой точке минимума, так что скорость обучения можно зафиксировать. Достаточные условия сходимости СГС имеют вид:

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \tag{8.12}$$

и

$$\sum_{k=1}^{\infty} \varepsilon_k^2 < \infty. \tag{8.13}$$

На практике скорость обучения обычно уменьшают линейно до итерации с номером τ:

$$\varepsilon_k = (1 - \alpha)\varepsilon_0 + \alpha\varepsilon_\tau, \tag{8.15}$$

где $\alpha = k/\tau$. После τ-й итерации ε остается постоянным.

Скорость обучения можно выбрать методом проб и ошибок, но обычно лучше понаблюдать за кривыми обучения – зависимостью целевой функции от времени. Здесь больше искусства, чем науки, поэтому большинство рекомендаций по этому вопросу следует воспринимать с долей скептицизма. Если скорость изменяется линейно, то нужно задать параметры ε_0, ε_τ и τ. Обычно в качестве τ выбирают число итераций, необходимое для выполнения нескольких сотен проходов по обучающему набору. Величину ε_τ задают равной примерно 1% от ε_0. Главный вопрос: как задать ε_0. Если значение слишком велико, то кривая обучения будет сильно осциллировать, а функция стоимости – значительно увеличиваться. Слабые осцилляции не несут угрозы, особенно если для обучения используется стохастическая функция стоимости, как, например, в случае прореживания. Если скорость обучения слишком мала, то обучение происходит медленно, а если слишком мала и начальная скорость, то обучение может застрять в точке с высокой стоимостью. Как правило, оптимальная начальная скорость обучения с точки зрения общего времени обучения и конечной стоимости выше, чем скорость, которая дает наилучшее качество после первых примерно 100 итераций. Поэтому обычно имеет смысл последить за первыми несколькими итерациями и взять скорость обучения большую, чем наилучшая на этом отрезке, но не настолько высокую, чтобы дело закончилось сильной неустойчивостью.

Самое важное свойство СГС и схожих методов мини-пакетной или онлайновой градиентной оптимизации заключается в том, что время вычислений в расчете на одно обновление не увеличивается с ростом числа обучающих примеров. Следовательно, сходимость возможна, даже когда число обучающих примеров очень велико. Если набор данных достаточно велик, то СГС может сойтись с некоторым фиксированным отклонением от финальной ошибки на тестовом наборе еще до завершения обработки всего обучающего набора.

Для изучения скорости сходимости алгоритма оптимизации часто измеряют **ошибку превышения** $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, т. е. величину, на которую текущая функция стоимости превышает минимально возможную стоимость. При применении СГС к выпуклой задаче ошибка превышения равна $O(1/\sqrt{k})$ после k итераций, тогда как в строго выпуклом случае она составляет $O(1/k)$. Эти границы нельзя улучшить без дополнительных предположений. Теоретически у пакетного градиентного спуска скорость сходимости должна быть выше, чем у стохастического. Но из неравенства Крамера–Рао (Cramér, 1946; Rao, 1945) следует, что ошибка обобщения не может убывать быстрее, чем $O(1/k)$. В работе Bottou and Bousquet (2008) утверждается, что в таком случае в задачах машинного обучения не имеет смысла искать алгоритм оптимизации, который сходится быстрее, чем $O(1/k)$, – более быстрая сходимость, скорее всего, приведет к переобучению. Кроме того, асимптотический анализ игнорирует многие преимущества стохастического градиентного спуска с небольшим числом шагов. На больших наборах способность СГС быстро достигать прогресса на начальной стадии после обсчета небольшого числа примеров перевешивает его медленную асимптотическую сходимость. Большинство описываемых далее алгоритмов обладает практически важными достоинствами, которые скрыты за постоянным множите-

лем в асимптотической оценке $O(1/k)$. Можно также попытаться найти компромисс между пакетным и стохастическим градиентным спусками, постепенно увеличивая размер мини-пакета в процессе обучения.

Дополнительные сведения о СГС см. в работе Bottou (1998).

8.3.2. Импульсный метод

Стохастический градиентный спуск остается популярной стратегией оптимизации, но обучение с его помощью иногда происходит слишком медленно. **Импульсный метод** (Polyak, 1964) призван ускорить обучение, особенно в условиях высокой кривизны, небольших, но устойчивых градиентов или зашумленных градиентов. В импульсном алгоритме вычисляется экспоненциально затухающее скользящее среднее прошлых градиентов и продолжается движение в этом направлении. Работа импульсного метода иллюстрируется на рис. 8.5.

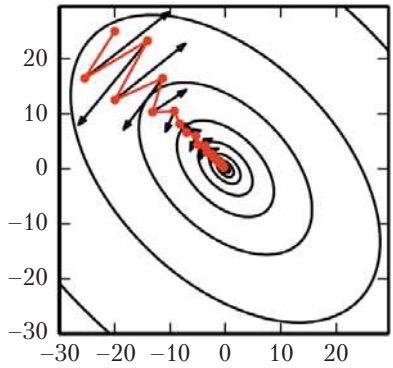

Рис. 8.5 ❖ Импульсный метод призван решить две проблемы: плохую обусловленность матрицы Гессе и дисперсию стохастического градиента. На рисунке показано, как преодолевается первая проблема. Эллипсы обозначают изолинии квадратичной функции потерь с плохо обусловленной матрицей Гессе. Красная линия, пересекающая эллипсы, соответствует траектории, выбираемой в соответствии с правилом обучения методом моментов в процессе минимизации этой функции. Для каждого шага обучения стрелка показывает, какое направление выбрал бы в этот момент метод градиентного спуска. Как видим, плохо обусловленная квадратичная целевая функция выглядит как длинная узкая долина или овраг с крутыми склонами. Импульсный метод правильно перемещается вдоль оврага, тогда как градиентный спуск впустую тратил бы время на перемещение вперед-назад поперек оврага. Сравните также с рис. 4.6, где показано поведение градиентного спуска без учета импульса

Формально говоря, в импульсном алгоритме вводится переменная $\boldsymbol{v}$, играющая роль скорости, – это направление и скорость перемещения в пространстве параметров. Скорость устанавливается равной экспоненциально затухающему скользящему среднему градиента со знаком минус. Название алгоритма проистекает из физической аналогии, согласно которой отрицательный градиент – это сила, под действием которой частица перемещается в пространстве параметров согласно законам Ньютона. В физике импульсом называется произведение массы на скорость. В импульс-

ном алгоритме масса предполагается единичной, поэтому вектор скорости $\boldsymbol{v}$ можно рассматривать как импульс частицы. Гиперпараметр $\alpha \in [0, 1)$ определяет скорость экспоненциального затухания вкладов предшествующих градиентов. Правило обновления имеет вид:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

Алгоритм 8.2. Стохастический градиентный спуск (СГС) с учетом импульса

Require: скорость обучения ε, параметр импульса α
Require: начальные значения параметров $\boldsymbol{\theta}$, начальная скорость $\boldsymbol{v}$
while критерий остановки не выполнен **do**
Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
Вычислить оценку градиента: $\boldsymbol{g} \leftarrow (1/m)\nabla_{\boldsymbol{\theta}} \Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Вычислить обновление скорости: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \boldsymbol{g}$.
Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
end while

В скорости $\boldsymbol{v}$ суммируются градиенты $\nabla_{\boldsymbol{\theta}}((1/m)\Sigma_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}))$. Чем больше α относительно ε, тем сильнее предшествующие градиенты влияют на выбор текущего направления. СГС с учетом импульса описан в алгоритме 8.2.

Раньше размер шага был равен просто норме градиента, умноженной на скорость обучения. Теперь же шаг зависит от величины и сонаправленности предшествующих градиентов. Размер шага максимален, когда много *последовательных* градиентов указывают точно в одном и том же направлении. Если импульсный алгоритм всегда видит градиент $\boldsymbol{g}$, то он будет ускоряться в направлении $-\boldsymbol{g}$, пока не достигнет конечной скорости, при которой размер шага равен

$$\frac{\varepsilon \| \boldsymbol{g} \|}{1 - \alpha}. \tag{8.17}$$

Таким образом, полезно рассматривать гиперпараметр импульса в терминах $1/(1 - \alpha)$. Например, $\alpha = 0.9$ соответствует умножению максимальной скорости на 10 относительно стандартного алгоритма градиентного спуска.

На практике обычно задают α равным 0.5, 0.9 или 0.99. Как и скорость обучения, α может меняться со временем. Как правило, начинают с небольшого значения и постепенно увеличивают его. Изменение α со временем не так важно, как уменьшение ε со временем.

Импульсный алгоритм можно рассматривать как имитацию движения частицы, подчиняющейся динамике Ньютона. Физическая аналогия помогает составить интуитивное представление о поведении алгоритма градиентного спуска и импульсного метода.

Положение частицы в любой момент времени описывается функцией $\boldsymbol{\theta}(t)$. К частице приложена суммарная сила $\boldsymbol{f}(t)$, под действием которой частица ускоряется:

$$\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2} \boldsymbol{\theta}(t). \tag{8.18}$$

Вместо того чтобы рассматривать это как дифференциальное уравнение второго порядка, описывающее положение частицы, мы можем ввести переменную $\boldsymbol{v}(t)$, представляющую скорость частицы в момент t, и выразить ньютоновскую динамику в виде уравнения первого порядка:

$$\boldsymbol{v}(t) = \frac{\partial}{\partial t}\boldsymbol{\theta}(t), \tag{8.19}$$

$$\boldsymbol{f}(t) = \frac{\partial}{\partial t}\boldsymbol{v}(t). \tag{8.20}$$

Тогда для применения импульсного алгоритма нужно численно решить эту систему дифференциальных уравнений. Простой способ решения дает метод Эйлера, который заключается в моделировании динамики, описываемой уравнением, путем небольших конечных шагов в направлении каждого градиента.

Итак, мы описали базовую форму обновления параметров импульсным методом, но что конкретно представляют собой силы? Одна сила пропорциональна отрицательному градиенту функции стоимости: $-\nabla_\theta J(\boldsymbol{\theta})$. Эта сила толкает частицу вниз по поверхности функции стоимости. Алгоритм градиентного спуска просто сделал бы один шаг, основанный на градиенте, но в импульсном алгоритме эта сила изменяет скорость частицы. Можно считать частицу хоккейной шайбой, скользящей по ледяной поверхности. Во время спуска по крутому склону она набирает скорость и продолжает скользить в одном направлении, пока не начнется очередной подъем.

Но необходима еще одна сила. Если бы единственной силой был градиент функции стоимости, то частица могла бы никогда не остановиться. Представьте себе шайбу, скользящую вниз по одному склону оврага, затем вверх по противоположному склону – если предположить, что трения нет, то она так и будет опускаться и подниматься бесконечно. Для решения этой проблемы мы добавим еще одну силу, пропорциональную $-\boldsymbol{v}(t)$. В физике мы назвали бы ее вязким сопротивлением, как если бы частица должна была прокладывать себе путь в сопротивляющейся среде, например в сиропе. В результате частица постепенно теряет энергию и в конце концов остановится в локальном минимуме.

Зачем использовать $-\boldsymbol{v}(t)$ и конкретно вязкое сопротивление? Отчасти из-за математического удобства – с целой степенью скорости проще работать. Но в других физических системах встречаются и иные виды сопротивления, основанные на целых степенях скорости. Например, частица, движущаяся в воздухе, испытывает турбулентное сопротивление, пропорциональное квадрату скорости, а частица, движущаяся по земле, – сопротивление трения силу постоянной величины. Но оба этих варианта следует отвергнуть. Турбулентное сопротивление, пропорциональное квадрату скорости, оказывается очень малым при малых скоростях. Его не хватит, чтобы заставить частицу остановиться. Частица с ненулевой начальной скоростью, на которую действует только сила турбулентного сопротивления, будет вечно удаляться от начальной точки, причем расстояние до нее растет как $O(\log t)$. Поэтому нужно брать меньшую степень скорости. Если взять степень 0, соответствующую сухому трению, то сила окажется слишком большой. Когда сила, обусловленная градиентом функции стоимости, мала, но все же отлична от нуля, постоянная сила трения может остановить частицу еще до достижения локального минимума. Сила вязкого сопротивления позволяет избежать обеих проблем – она достаточно слаба, чтобы градиент

продолжал вызывать движение до достижения минимума, и достаточна сильна, чтобы предотвратить движение, не оправдываемое градиентом.

8.3.3. Метод Нестерова

В работе Sutskever et al. (2013) описан вариант импульсного алгоритма, созданный под влиянием метода ускоренного градиента Нестерова (Nesterov, 1983, 2004). Правила обновления в этом случае имеют вид:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)}) \right], \tag{8.21}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \tag{8.22}$$

Алгоритм 8.3. Стохастический градиентный спуск (СГС) методом Нестерова

Require: скорость обучения ε, параметр импульса α
Require: начальные значения параметров $\boldsymbol{\theta}$, начальная скорость $\boldsymbol{v}$
while критерий остановки не выполнен **do**
Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
Выполнить промежуточное обновление: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Вычислить градиент (в промежуточной точке): $\boldsymbol{g} \leftarrow (1/m) \nabla_{\tilde{\theta}} \Sigma_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.
Вычислить обновление скорости: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \boldsymbol{g}$
Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
end while

где параметры α и ε играют ту же роль, что в стандартном импульсном методе. Разница между методом Нестерова и стандартным импульсным методом заключается в точке, где вычисляется градиент. В методе Нестерова градиент вычисляется после применения текущей скорости. Таким образом, метод Нестерова можно интерпретировать как попытку добавить поправочный множитель к стандартному импульсному методу. Полностью метод Нестерова представлен в алгоритме 8.3.

В случае выпуклой оптимизации пакетным градиентным спуском метод Нестерова повышает скорость сходимости ошибки превышения с $O(1/k)$ (после k шагов) до $O(1/k^2)$, как доказано в работе Nesterov (1983). К сожалению, в случае стохастического градиентного спуска метод Нестерова не улучшает скорости сходимости.

8.4. Стратегии инициализации параметров

Некоторые алгоритмы оптимизации по природе своей не итеративные, они просто решают уравнение. Другие алгоритмы хоть и итеративные, но в применении к подходящему классу задач оптимизации сходятся к приемлемому решению за приемлемое время вне зависимости от начальных значений. Алгоритмы глубокого обучения обычно не обладают столь приятными свойствами. Как правило, они итеративные, и потому пользователь должен указать, с какой точки начинать итерации. Кроме того, обучение глубоких моделей – задача настолько сложная, что большинство алгоритмов сильно зависит от выбора начальных значений. От начальной точки может зависеть, сойдется ли вообще алгоритм, причем некоторые начальные точки так

неустойчивы, что алгоритм сталкивается с численными трудностями и завершается ошибкой. Если все-таки алгоритм сходится, то начальная точка определяет скорость сходимости и стоимость в конечной точке – высокую или низкую. Кроме того, для точек с сопоставимой стоимостью ошибка обобщения может различаться очень сильно, и на нее начальная точка тоже может оказывать влияние.

Современные стратегии инициализации просты и основаны на эвристических соображениях. Проектирование улучшенной стратегии инициализации – трудная задача, потому что еще нет отчетливого понимания оптимизации нейронных сетей. Большинство стратегий основано на стремлении получить некоторые полезные свойства сети в начальный момент. Однако мы плохо понимаем, какие из этих свойств и при каких условиях сохраняются после начала процесса обучения. Дополнительная трудность состоит в том, что некоторые начальные точки хороши с точки зрения оптимизации, но никуда не годятся с точки зрения обобщаемости. Наше понимание того, как начальная точка влияет на обобщаемость, совсем уж примитивно, оно не дает почти или вообще никаких указаний на то, как выбирать начальную точку.

Пожалуй, единственное, что мы знаем наверняка, – это то, что начальные параметры должны «нарушить симметрию» между разными блоками. Если два скрытых блока с одинаковыми функциями активации соединены с одними и теми же входами, то у этих блоков должны быть разные начальные параметры. Если начальные параметры одинаковы, то детерминированный алгоритм обучения, применяемый к детерминированной функции стоимости и модели, будет всякий раз обновлять эти блоки одинаково. Даже если модель или алгоритм обучения способны стохастически вычислять разные обновления разных блоков (например, если при обучении используется прореживание), обычно предпочтительнее инициализировать каждый блок, так чтобы он вычислял свою функцию иначе, чем все остальные блоки. Это позволит гарантировать, что никакие входные паттерны не потеряются в нуль-пространстве прямого распространения, и никакие паттерны градиентов не потеряются в нуль-пространстве обратного распространения. Стремление к тому, чтобы все блоки вычисляли разные функции, диктует выбор в пользу случайной инициализации параметров. Мы могли бы провести явный поиск в большом множестве взаимно различных базисных функций, но это зачастую дорого обходится с точки зрения вычислений. Например, если число выходов не превышает число входов, то можно было бы воспользоваться ортогонализацией Грама–Шмидта матрицы начальных весов и тем самым гарантировать, что все блоки будут вычислять очень сильно различающиеся функции. Случайная инициализация на основе распределения с высокой энтропией в многомерном пространстве вычислительно дешевле, а вероятность назначить двум блокам одинаковую функцию крайне мала.

В типичном случае мы выбираем в качестве смещений блоков эвристически выбранные константы, а случайно инициализируем только веса. Дополнительные параметры, например условная дисперсия предсказания, обычно тоже задаются эвристическими константами.

Почти всегда веса модели инициализируются случайными значениями с нормальным или равномерным распределением. Вопрос о том, какое распределение лучше, похоже, не играет особой роли, но тщательно он не изучался. А вот масштаб начального распределения сильно влияет как на результат процедуры оптимизации, так и на способность сети к обобщению.

Чем больше начальные веса, тем сильнее эффект нарушения симметрии, что помогает избежать избыточных блоков. Большие начальные веса помогают также пре-

дотвратить потерю сигнала во время прямого или обратного распространения через линейные компоненты каждого слоя – чем больше значения в матрице, тем больше результат умножения матриц. Однако если начальные веса слишком велики, то может случиться взрывной рост значений во время прямого или обратного распространения. В рекуррентных сетях большие веса также могут привести к **хаосу** (настолько высокой чувствительности к малым возмущениям входного сигнала, что поведение детерминированной процедуры прямого распространения представляется случайным). Проблему взрывного градиента можно в какой-то мере сгладить путем отсечения градиента (сравнения градиента с порогом перед выполнением шага градиентного спуска). Кроме того, большие веса могут стать причиной экстремальных значений, что ведет к насыщению функции активации и полной потере градиента при распространении через насыщенные блоки. Балансирование этих разнонаправленных факторов и определяет идеальный масштаб весов.

Взгляды на проблему с точки зрения регуляризации и оптимизации могут дать совершенно разные подходы к инициализации сети. С точки зрения оптимизации, веса должны быть достаточно большими, чтобы способствовать успешному распространению информации. Но соображения регуляризации побуждают делать веса поменьше. Использование таких алгоритмов оптимизации, как стохастический градиентный спуск, который производит инкрементные изменения весов и выказывает тенденцию к остановке в областях, близких к начальным параметрам (то ли из-за застревания в области низких градиентов, то ли потому, что сработал критерий ранней остановки вследствие угрозы переобучения), выражает априорное знание о том, что конечные параметры должны быть близки к начальным. Напомним (см. раздел 7.8), что для некоторых моделей градиентный спуск с ранней остановкой эквивалентен снижению весов. В общем случае градиентный спуск с ранней остановкой – не то же самое, что снижение весов, но между ними можно провести некоторую аналогию, позволяющую рассуждать об эффекте инициализации. Мы можем считать инициализацию параметров $\boldsymbol{\theta}$ значениями $\boldsymbol{\theta}_0$ аналогом гипотезы о нормальном априорном распределении $p(\boldsymbol{\theta})$ со средним $\boldsymbol{\theta}_0$. С этой точки зрения, имеет смысл выбирать $\boldsymbol{\theta}_0$ близким к 0. При таком априорном распределении отсутствие взаимодействия между блоками вероятнее его наличия. Блоки взаимодействуют, только если член правдоподобия в целевой функции выражает сильное предпочтение к такому взаимодействию. С другой стороны, если инициализировать $\boldsymbol{\theta}_0$ большими значениями, то априорная гипотеза говорит о том, какие блоки должны взаимодействовать между собой и как именно.

Существуют некоторые эвристики для выбора начального масштаба весов. Одна из них – инициализировать веса в полносвязном слое с m входами и n выходами, выбирая каждый вес из распределения $U(-1/\sqrt{m}, 1/\sqrt{m})$. А в работе Glorot and Bengio (2010) предлагается использовать **нормированную инициализацию**

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{8.23}$$

Эта последняя эвристика выражает компромисс между желанием инициализировать все слои, так чтобы была одинакова дисперсия активации, и желанием инициализировать их, так чтобы была одинаковая дисперсия градиента. Формула выведена в предположении, что сеть включает только цепочку умножений матриц безо всяких нелинейностей. Очевидно, что реальные нейронные сети не удовлетворяют этому

предположению, но многие стратегии, разработанные для линейных моделей, дают неплохие результаты и в нелинейных сетях.

В работе Saxe et al. (2013) рекомендуется в качестве начальных значений брать случайные ортогональные матрицы с тщательно подобранным масштабным коэффициентом **усиления** g, который учитывает нелинейности в каждом слое. Авторы приводят конкретные значения масштабного коэффициента для нелинейных функций активации разных типов. Обоснованием такой схемы инициализации также служит модель глубокой сети как последовательности умножений матриц без нелинейностей. При такой модели схема гарантирует, что общее число итераций обучения, необходимое для достижения сходимости, не зависит от глубины.

Увеличение масштабного коэффициента g переводит сеть в режим, когда норма активации возрастает при прямом распространении, а норма градиента – при обратном. В работе Sussillo (2014) показано, что правильного выбора коэффициента усиления достаточно для обучения глубоких сетей с 1000 уровней без применения ортогональной инициализации. Главная идея этого подхода состоит в том, что в сетях прямого распространения активация и градиент могут возрастать или убывать на каждом шаге прямого или обратного распространения, как при случайном блуждании. Объясняется это тем, что в сетях прямого распространения в каждом слое используется своя матрица весов. Если настроить это случайное блуждание, так чтобы норма сохранялась, то сеть прямого распространения сможет в большинстве случаев избежать проблемы исчезающих и взрывных градиентов, которая возникает, когда на каждом шаге используется одна и та же матрица (см. раздел 8.2.5).

К сожалению, оптимальные критерии для начальных весов зачастую не приводят к оптимальному качеству. Тому может быть три причины. Во-первых, неподходящий критерий – возможно, он не способствует сохранению нормы сигнала во всей сети. Во-вторых, свойства, справедливые в момент инициализации, могут нарушаться после начала обучения. В-третьих, критерий может ускорять оптимизацию, но непреднамеренно увеличивать ошибку обобщения. На практике масштаб весов обычно следует рассматривать как гиперпараметр, оптимальное значение которого близко к теоретически предсказанному, но не совпадает с ним.

Недостаток правил масштабирования, при которых все начальные веса имеют одинаковое стандартное отклонение, например $1/\sqrt{m}$, состоит в том, что каждый отдельный вес становится очень малым, когда число слоев растет. В работе Martens (2010) предложена альтернативная схема, названная **разреженной инициализацией**, при которой каждому блоку в начальный момент назначается ровно k ненулевых весов. Идея в том, чтобы сделать общий объем входящей в блок информации независимым от числа входов m, не заставляя абсолютную величину отдельных весовых элементов уменьшаться вместе с m. Разреженная инициализация помогает обеспечить большее разнообразие блоков на стадии инициализации. Однако она также предполагает очень сильные априорные предположения о весах, которые должны иметь большие нормально распределенные значения. Поскольку градиентному спуску понадобится много времени, чтобы сократить «неправильные» большие значения, эта схема инициализации может вызывать проблемы для таких блоков (в частности, maxout-блоков), у которых есть несколько фильтров, которые должны быть тщательно скоординированы друг с другом.

Если вычислительные ресурсы позволяют, обычно имеет смысл рассматривать начальный масштаб весов в каждом слое как гиперпараметр и выбирать масштабы,

применяя какой-нибудь из алгоритмов поиска гиперпараметров, описанных в разделе 11.4.2, например случайный поиск. Или можно вручную поискать наилучшие возможные веса. Хорошее эвристическое правило выбора начальных масштабов – проанализировать диапазон стандартного отклонения активаций или градиентов на одном мини-пакете данных. Если веса слишком малы, то диапазон активаций будет сужаться по мере прямого распространения по сети. Раз за разом определяя первый слой с неприемлемой малой активацией и увеличивая веса в нем, можно в конце концов получить сеть с разумными начальными активациями снизу доверху. Если в этом момент обучение все еще происходит слишком медленно, то полезно также проанализировать диапазон стандартных отклонений градиентов и активаций. В принципе, эту процедуру можно автоматизировать, и в общем случае она вычислительно дешевле, чем оптимизация гиперпараметра, основанная на ошибке на контрольном наборе, поскольку в основе лежит обратная связь с поведением начальной модели на одном пакете данных, а не с обученной моделью на контрольном наборе. Этот эвристический протокол используется уже долгое время, но лишь недавно он был формализован и изучен в работе Mishkin and Matas (2015).

До сих пор мы говорили об инициализации весов. К счастью, инициализация других параметров обычно проще.

Подходы к заданию смещений и весов должны быть согласованы. Инициализация всех смещений нулями совместима с большинством схем инициализации весов. Есть несколько ситуаций, в которых разумно присваивать некоторым смещениям ненулевые значения.

- Если речь идет о смещении для выходного блока, то часто имеет смысл инициализировать его так, чтобы получилась правильная маргинальная статистика выхода. Для этого предположим, что начальные веса настолько малы, что выход блока определяется только смещением. Это оправдывает выбор в качестве смещения величины, обратной значению функции активации, примененной к маргинальной статистике выхода в обучающем наборе. Например, если выходом является распределение классов, и это распределение сильно скошено, а маргинальная вероятность i-го класса задается элементом c_i некоторого вектора $\boldsymbol{c}$, то вектор смещений $\boldsymbol{b}$ можно найти из уравнения $\text{softmax}(\boldsymbol{b}) = \boldsymbol{c}$. Это относится не только к классификаторам, но и к моделям, с которыми мы встретимся в части III, например автокодировщикам и машинам Больцмана. В этих моделях имеются слои, выход которых должен быть похож на вход $\boldsymbol{x}$, и было бы очень полезно инициализировать смещения таких слоев в соответствии с маргинальным распределением $\boldsymbol{x}$.
- Иногда мы хотим выбрать смещение так, чтобы предотвратить слишком сильное насыщение на стадии инициализации. Например, мы можем задать смещение скрытого ReLU-блока равным 0.1, а не 0, чтобы избежать его насыщения. Но этот подход несовместим со схемами инициализации весов, которые не ожидают сильного входного сигнала от смещений. Например, его не рекомендуется использовать вместе с инициализацией случайным блужданием (Sussillo, 2014).
- Иногда один блок управляет тем, могут ли другие блоки принимать участие в вычислении функции. В таких ситуациях имеются блок с выходом u и другой блок $h \in [0, 1]$, они перемножаются, и на выходе получается uh. Мы можем рассматривать h как вентиль, определяющий, будет ли $uh \approx u$ или $uh \approx 0$. Тогда мы

хотим на этапе инициализации задать смещение для h, так чтобы $h \approx 1$ большую часть времени. В противном случае у u не будет возможности обучиться. Например, в работе Jozefowicz et al. (2015) рекомендуется устанавливать смещение 1 для вентиля забывания в модели LSTM, описанной в разделе 10.10.

Еще один распространенный параметр – дисперсия, или точность. Например, мы можем выполнить линейную регрессию с оценкой условной дисперсии с помощью модели

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^T\boldsymbol{x} + b, 1/\beta), \tag{8.24}$$

где β – параметр точности. Обычно безопасно инициализировать дисперсию, или точность, значением 1. Другой подход – предположить, что начальные веса настолько близки к нулю, что смещения можно задавать, игнорируя влияние весов, и тогда задать смещения так, чтобы порождалось правильное маргинальное среднее выхода, а дисперсии сделать равными маргинальной дисперсии выхода в обучающем наборе.

Помимо простых методов инициализации параметров модели постоянными или случайными значениями, можно для этой цели применить машинное обучение. Типичная стратегия, обсуждаемая в части III, – инициализировать модель с учителем параметрами, обученными с помощью модели без учителя на тех же входных данных. Можно также выполнить обучение с учителем на родственной задаче. Даже обучение с учителем на никак не связанной задаче может иногда дать начальные значения, обеспечивающие более быструю, по сравнению со случайной инициализацией, сходимость. Некоторые стратегии инициализации такого рода могут приводить к ускоренной сходимости и лучшей обобщаемости, потому что в них закодирована информация о распределении начальных параметров модели. Другие дают хорошие результаты, по всей видимости, из-за того, что выбирают правильный масштаб параметров или настраивают блоки на вычисление различных функций.

8.5. Алгоритмы с адаптивной скоростью обучения

Специалисты по нейронным сетям давно поняли, что скорость обучения – один из самых трудных для установки гиперпараметров, поскольку она существенно влияет на качество модели. В разделах 4.3 и 8.2 мы говорили о том, что стоимость зачастую очень чувствительна в некоторых направлениях пространства параметров и нечувствительна в других. Импульсный алгоритм может в какой-то мере сгладить эти проблемы, но ценой введения другого гиперпараметра. Естественно возникает вопрос, нет ли какого-то иного способа. Если мы полагаем, что направления чувствительности почти параллельны осям, то, возможно, имеет смысл задавать скорость обучения отдельно для каждого параметра и автоматически адаптировать эти скорости на протяжении всего обучения.

Алгоритм **delta-bar-delta** (Jacobs, 1988) – один из первых эвристических подходов к адаптации индивидуальных скоростей обучения параметров модели. Он основан на простой идее: если частная производная функции потерь по данному параметру модели не меняет знак, то скорость обучения следует увеличить. Если же знак меняется, то скорость следует уменьшить. Конечно, такого рода правило применимо только к оптимизации на полном пакете.

Позже был предложен целый ряд инкрементных (или основанных на мини-пакетах) методов для адаптации скоростей обучения параметров. В этом разделе мы кратко рассмотрим некоторые из них.

8.5.1. AdaGrad

Алгоритм **AdaGrad** (алгоритм 8.4) по отдельности адаптирует скорости обучения всех параметров модели, умножая их на коэффициент, обратно пропорциональный квадратному корню из суммы всех прошлых значений квадрата градиента (Duchi et al., 2011). Для параметров, по которым частная производная функции потерь наибольшая, скорость обучения уменьшается быстро, а если частная производная мала, то и скорость обучения уменьшается медленнее. В итоге больший прогресс получается в направлениях пространства параметров со сравнительно пологими склонами.

В случае выпуклой оптимизации у алгоритма AdaGrad есть некоторые желательные теоретические свойства. Но эмпирически при обучении глубоких нейронных сетей накапливание квадратов градиента с самого начала обучения может привести к преждевременному и чрезмерному уменьшению эффективной скорости обучения. AdaGrad хорошо работает для некоторых, но не для всех моделей глубокого обучения.

8.5.2. RMSProp

Алгоритм RMSProp (Hinton, 2012) – это модификация AdaGrad, призванная улучшить его поведение в невыпуклом случае путем изменения способа агрегирования градиента на экспоненциально взвешенное скользящее среднее. AdaGrad разрабатывался для быстрой сходимости в применении к выпуклой функции. Если же он применяется к невыпуклой функции для обучения нейронной сети, то траектория обучения может проходить через много разных структур и в конечном итоге прийти в локально выпуклую впадину. AdaGrad уменьшает скорость обучения, принимая во внимание всю историю квадрата градиента, и может случиться так, что скорость станет слишком малой еще до достижения такой выпуклой структуры. В алгоритме RMSProp используется экспоненциально затухающее среднее, т. е. далекое прошлое отбрасывается, чтобы повысить скорость сходимости после обнаружения выпуклой впадины, как если бы внутри этой впадины алгоритм AdaGrad был инициализирован заново.

Алгоритм 8.4. Алгоритм AdaGrad

Require: глобальная скорость обучения ε
Require: начальные значения параметров $\boldsymbol{\theta}$
Require: небольшая константа δ, например 10^{-7}, для обеспечения численной устойчивости
Инициализировать переменную для агрегирования градиента $\boldsymbol{r} = \mathbf{0}$
while критерий остановки не выполнен **do**
Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
Вычислить градиент: $\boldsymbol{g} \leftarrow (1/m)\nabla_{\theta}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Агрегировать квадраты градиента: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
Вычислить обновление: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\varepsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$ (операции деления и извлечения корня применяются к каждому элементу).
Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

Алгоритм 8.5 содержит описание RMSProp в стандартной форме, а алгоритм 8.6 – в сочетании с методом Нестерова. По сравнению с AdaGrad вводится новый гиперпараметр ρ, управляющий масштабом длины при вычислении скользящего среднего.

Алгоритм 8.5. Алгоритм RMSProp

Require: глобальная скорость обучения ε, скорость затухания ρ
Require: начальные значения параметров $\boldsymbol{\theta}$
Require: небольшая константа δ, например 10^{-6}, для стабилизации деления на малые числа
Инициализировать переменную для агрегирования градиента $\boldsymbol{r} = 0$
while критерий остановки не выполнен **do**
Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
Вычислить градиент: $\boldsymbol{g} \leftarrow (1/m)\nabla_{\theta}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Агрегировать квадраты градиента: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1 - \rho)\, \boldsymbol{g} \odot \boldsymbol{g}$.
Вычислить обновление параметров: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\varepsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$ (операция $\frac{\varepsilon}{\sqrt{\delta + \boldsymbol{r}}}$ применяется к каждому элементу).
Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

Алгоритм 8.6. Алгоритм RMSProp в сочетании с методом Нестерова

Require: глобальная скорость обучения ε, скорость затухания ρ, коэффициент импульса α
Require: начальные значения параметров $\boldsymbol{\theta}$, начальная скорость $\boldsymbol{v}$
Инициализировать переменную для агрегирования градиента $\boldsymbol{r} = \boldsymbol{0}$
while критерий остановки не выполнен **do**
Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
Вычислить промежуточное обновление: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{v}$
Вычислить градиент: $\boldsymbol{g} \leftarrow (1/m)\nabla_{\tilde{\theta}}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.
Агрегировать градиент: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1 - \rho)\, \boldsymbol{g} \odot \boldsymbol{g}$.
Вычислить обновление скорости: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - (\varepsilon/\sqrt{\boldsymbol{r}}) \odot \boldsymbol{g}$ (операция $1/\sqrt{\boldsymbol{r}}$ применяется к каждому элементу).
Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
end while

Эмпирически показано, что RMSProp – эффективный и практичный алгоритм оптимизации глубоких нейронных сетей. В настоящее время он считается одним из лучших методов оптимизации и постоянно используется в практической работе.

8.5.3. Adam

Adam (Kingma and Ba, 2014) – еще один алгоритм оптимизации с адаптивной скоростью обучения – описан в алгоритме 8.7. Название «Adam» – сокращение от «adaptive moments» (адаптивные моменты). Его, наверное, правильнее всего рассматривать как комбинацию RMSProp и импульсного метода с несколькими важными отличиями. Во-первых, в Adam импульс включен непосредственно в виде оценки первого момента (с экспоненциальными весами) градиента. Самый прямой способ добавить

импульс в RMSProp – применить его к масштабированным градиентам. У использования импульса в сочетании с масштабированием нет ясного теоретического обоснования. Во-вторых, Adam включает поправку на смещение в оценки как первых моментов (член импульса), так и вторых (нецентрированных) моментов для учета их инициализации в начале координат (см. алгоритм 8.7). RMSProp также включает оценку (нецентрированного) второго момента, однако в нем нет поправочного коэффициента. Таким образом, в отличие от Adam, в RMSProp оценка второго момента может иметь высокое смещение на ранних стадиях обучения. Вообще говоря, Adam считается довольно устойчивым к выбору гиперпараметров, хотя скорость обучения иногда нужно брать отличной от предлагаемой по умолчанию.

Алгоритм 8.7. Алгоритм Adam

Require: величина шага ε (по умолчанию 0.001).
Require: коэффициенты экспоненциального затухания для оценок моментов ρ_1 и ρ_2, принадлежащие диапазону [0, 1) (по умолчанию 0.9 и 0.999 соответственно).
Require: небольшая константа δ для обеспечения численной устойчивости (по умолчанию 10^{-8}).
Require: начальные значения параметров $\boldsymbol{\theta}$.
Инициализировать переменные для первого и второго моментов $\boldsymbol{s} = \mathbf{0}, \boldsymbol{r} = \mathbf{0}$
Инициализировать шаг по времени $t = 0$
while критерий остановки не выполнен **do**
Выбрать из обучающего набора мини-пакет m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ и соответствующие им метки $\boldsymbol{y}^{(i)}$.
Вычислить градиент: $\boldsymbol{g} \leftarrow (1/m)\nabla_{\boldsymbol{\theta}}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
$t \leftarrow t + 1$
Обновить смещенную оценку первого момента: $\boldsymbol{s} \leftarrow \rho_1\boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Обновить смещенную оценку второго момента: $\boldsymbol{r} \leftarrow \rho_2\boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Скорректировать смещение первого момента: $\hat{\boldsymbol{s}} \leftarrow \dfrac{\boldsymbol{s}}{1 - \rho_1^t}$
Скорректировать смещение второго момента: $\hat{\boldsymbol{r}} \leftarrow \dfrac{\boldsymbol{r}}{1 - \rho_2^t}$
Вычислить обновление: $\Delta\boldsymbol{\theta} = -\varepsilon \dfrac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (операции применяются к каждому элементу)
Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

8.5.4. Выбор правильного алгоритма оптимизации

Мы обсудили ряд родственных алгоритмов, каждый из которых пытается решить проблему оптимизации глубоких моделей, адаптируя скорость обучения каждого параметра. Возникает естественный вопрос: какой алгоритм выбрать?

К сожалению, в настоящее время единого мнения нет. В работе Schaul et al. (2014) представлено ценное сравнение большого числа алгоритмов оптимизации в применении к различным задачам обучения. И хотя результаты показывают, что семейство алгоритмов с адаптивной скоростью обучения (представленное алгоритмами RMSProp и AdaDelta) ведет себя достаточно устойчиво, явный победитель не выявлен.

Сейчас наиболее популярны и активно применяются алгоритмы СГС, СГС с учетом импульса, RMSProp, RMSProp с учетом импульса, AdaDelta и Adam. Какой из них использовать, зависит главным образом от знакомства пользователя с алгоритмом (читай: умения настраивать гиперпараметры).

8.6. Приближенные методы второго порядка

В этом разделе мы обсудим применение методов второго порядка к обучению глубоких сетей. Одно из первых изложений этой темы см. в работе LeCun et al. (1998a). Для простоты мы будем рассматривать только одну целевую функцию: эмпирический риск:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)}). \tag{8.25}$$

Впрочем, рассматриваемые здесь методы легко обобщаются на другие целевые функции, в т. ч. включающие члены регуляризации, обсуждавшиеся в главе 7.

8.6.1. Метод Ньютона

В разделе 4.3 мы познакомились с градиентными методами второго порядка. В отличие от методов первого порядка, в этом случае для улучшения оптимизации задействуются вторые производные. Самый известный метод второго порядка – метод Ньютона. Опишем его более подробно с акцентом на применении к обучению нейронных сетей.

Метод Ньютона основан на использовании разложения в ряд Тейлора с точностью до членов второго порядка для аппроксимации $J(\boldsymbol{\theta})$ в окрестности некоторой точки $\boldsymbol{\theta}_0$, производные более высокого порядка при этом игнорируются.

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta}-\boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + {}^1\!/_2 (\boldsymbol{\theta}-\boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta}-\boldsymbol{\theta}_0), \tag{8.26}$$

где $\boldsymbol{H}$ – гессиан J относительно $\boldsymbol{\theta}$, вычисленный в точке $\boldsymbol{\theta}_0$. Пытаясь найти критическую точку этой функции, мы приходим к правилу Ньютона для обновления параметров:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \tag{8.27}$$

Таким образом, для локально квадратичной функции (с положительно определенной матрицей $\boldsymbol{H}$) умножение градиента на $\boldsymbol{H}^{-1}$ сразу дает точку минимума. Если целевая функция выпуклая, но не квадратичная (имеются члены более высокого порядка), то это обновление можно повторить, получив тем самым алгоритм обучения 8.8, основанный на методе Ньютона.

Для неквадратичных поверхностей метод Ньютона можно применять итеративно, при условии что матрица Гессе остается положительно определенной. Отсюда вытекает двухшаговая итеративная процедура. Сначала мы обновляем или вычисляем обратный гессиан (путем обновления квадратичной аппроксимации). Затем обновляем параметры в соответствии с формулой (8.27).

Алгоритм 8.8. Метод Ньютона с целевой функций $J(\boldsymbol{\theta}) = (1/m)\Sigma_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)})$

Require: начальные значения параметров $\boldsymbol{\theta}_0$.
Require: обучающий набор m примеров

while критерий остановки не выполнен **do**
  Вычислить градиент: $\boldsymbol{g} \leftarrow (1/m)\nabla_{\boldsymbol{\theta}}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Вычислить гессиан: $\boldsymbol{H} \leftarrow (1/m)\nabla^2_{\boldsymbol{\theta}}\Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Вычислить обратный гессиан: $\boldsymbol{H}^{-1}$
  Вычислить обновление: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
  Применить обновление: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

В разделе 8.2.3 мы говорили, что метод Ньютона применим, только если матрица Гессе положительно определенная. В глубоком обучении поверхность целевой функции обычно невыпуклая и имеет много особенностей типа седловых точек, с которыми метод Ньютона не справляется. Если не все собственные значения гессиана положительны, например вблизи седловой точки, то метод Ньютона может произвести обновление не в том направлении. Такую ситуацию можно предотвратить с помощью регуляризации гессиана. Одна из распространенных стратегий регуляризации – прибавление константы α ко всем диагональным элементам гессиана. Тогда регуляризированное обновление принимает вид:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [\boldsymbol{H}(f(\boldsymbol{\theta}_0)) + \alpha\boldsymbol{I}]^{-1}\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0). \tag{8.27}$$

Эта стратегия регуляризации применяется в аппроксимациях метода Ньютона, например в алгоритме Левенберга–Марквардта (Levenberg, 1944; Marquardt, 1963), и работает неплохо, если отрицательные собственные значения гессиана сравнительно близки к нулю. Если же в некоторых направлениях кривизна сильнее, то значение α следует выбирать достаточно большим для компенсации отрицательных собственных значений. Однако по мере увеличения α в гессиане начинает доминировать диагональ $\alpha\boldsymbol{I}$, и направление, выбранное методом Ньютона, стремится к стандартному градиенту, поделенному на α. При наличии сильной кривизны α должно быть настолько большим, чтобы метод Ньютона делал меньшие шаги, чем градиентный спуск с подходящей скоростью обучения.

Помимо проблем, связанных с такими особенностями целевой функции, как седловые точки, применение метода Ньютона к обучению больших нейронных сетей лимитируется требованиями к вычислительным ресурсам. Число элементов гессиана равно квадрату числа параметров, поэтому при k параметрах (а даже для совсем небольшой нейронной сети параметры могут исчисляться миллионами) метод Ньютона требует обращения матрицы размера $k \times k$, а вычислительная сложность этой операции составляет $O(k^3)$. Кроме того, поскольку параметры изменяются при каждом обновлении, обратный гессиан нужно вычислять на каждой итерации обучения. Поэтому лишь сети с очень небольшим числом параметров реально обучить методом Ньютона. Далее в этом разделе мы обсудим альтернативы, смысл которых – воспользоваться некоторыми достоинствами метода Ньютона, не взваливая на себя неподъемного груза вычислений.

8.6.2. Метод сопряженных градиентов

Метод сопряженных градиентов позволяет избежать вычисления обратного гессиана посредством итеративного спуска в **сопряженных направлениях**. Идея этого подхода вытекает из внимательного изучения слабого места метода наискорейшего спуска (детали см. в разделе 4.3), при котором поиск итеративно производится в направле-

нии градиента. На рис. 8.6 показано, что метод наискорейшего спуска в квадратичной впадине неэффективен, т. к. продвигается зигзагами. Так происходит, потому что направление линейного поиска, определяемое градиентом на очередном шаге, гарантированно ортогонально направлению поиска на предыдущем шаге.

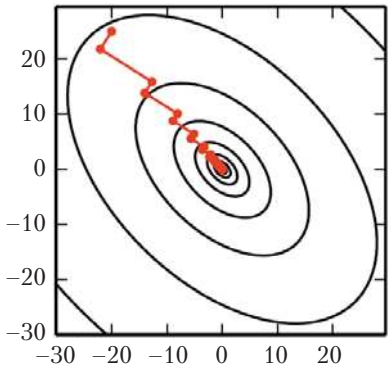

Рис. 8.6 ❖ Метод наискорейшего спуска в применении к поверхности квадратичной целевой функции. В этом методе на каждом шаге производится переход в точку с наименьшей стоимостью вдоль прямой, определяемой градиентом в начале этого шага. Это решает некоторые показанные на рис. 4.6 проблемы, которые обусловлены фиксированной скоростью обучения, но даже при оптимальной величине шага алгоритм все равно продвигается к оптимуму зигзагами. По определению, в точке минимума целевой функции вдоль заданного направления градиент в конечной точке ортогонален этому направлению

Обозначим $\boldsymbol{d}_{t-1}$ направление предыдущего поиска. В точке минимума, где поиск завершается, производная по направлению $\boldsymbol{d}_{t-1}$ равна нулю: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \boldsymbol{d}_{t-1} = 0$. Поскольку градиент в этой точке определяет текущее направление поиска, $\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ не дает вклада в направлении $\boldsymbol{d}_{t-1}$. Следовательно, $\boldsymbol{d}_t$ ортогонально $\boldsymbol{d}_{t-1}$. Эта связь между $\boldsymbol{d}_{t-1}$ и $\boldsymbol{d}_t$ иллюстрируется на рис. 8.6 для нескольких итераций метода наискорейшего спуска. Как видно по рисунку, выбор ортогональных направлений спуска не сохраняет минимума вдоль предыдущих направлений поиска. Отсюда и зигзагообразная траектория, поскольку после спуска к минимуму в направлении текущего градиента мы должны заново минимизировать целевую функцию в направлении предыдущего градиента. А значит, следуя направлению градиента в конце каждого отрезка, мы в некотором смысле перечеркиваем достигнутое в направлении предыдущего отрезка. Метод сопряженных градиентов и призван решить эту проблему.

В методе сопряженных градиентов направление следующего поиска является сопряженным к направлению предыдущего, т. е. мы не отказываемся от того, что было достигнуто в предыдущем направлении. На t-ой итерации обучения направление следующего поиска определяется формулой:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}, \tag{8.29}$$

где β_t – коэффициент, определяющий, какую часть направления $\boldsymbol{d}_{t-1}$ следует прибавить к текущему направлению поиска.

Два направления $\boldsymbol{d}_t$ и $\boldsymbol{d}_{t-1}$ называются сопряженными, если $\boldsymbol{d}_t^\top \boldsymbol{H}\boldsymbol{d}_{t-1} = 0$, где $\boldsymbol{H}$ – матрица Гессе.

Самый простой способ обеспечить сопряженность – вычислить собственные векторы $\boldsymbol{H}$ для выбора β_t – не отвечает исходной цели разработать метод, который был бы вычислительно проще метода Ньютона при решении больших задач. Можно ли найти сопряженные направления, не прибегая к таким вычислениям? К счастью, да.

Существуют два популярных метода вычисления β_t.

1. Метод Флетчера-Ривса:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}. \tag{8.30}$$

2. Метод Полака-Рибьера:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}. \tag{8.31}$$

Для квадратичной поверхности сопряженность направлений гарантирует, что модуль градиента вдоль предыдущего направления не увеличится. Поэтому минимум, найденный вдоль предыдущих направлений, сохраняется. Следовательно, в k-мерном пространстве параметров метод сопряженных градиентов требует не более k поисков для нахождения минимума. Этот метод описан в алгоритме 8.9.

Нелинейный метод сопряженных градиентов. До сих пор мы обсуждали метод сопряженных градиентов в применении к квадратичной целевой функции. Но в этой главе нас интересуют в основном методы оптимизации для обучения нейронных сетей и других глубоких моделей, в которых целевая функция далека от квадратичной. Как ни странно, метод сопряженных градиентов применим и в такой ситуации, хотя и с некоторыми изменениями. Если целевая функция не квадратичная, то уже нельзя гарантировать, что поиск в сопряженном направлении сохраняет минимум в предыдущих направлениях. Поэтому в нелинейном алгоритме сопряженных градиентов время от времени производится сброс, когда метод сопряженных градиентов заново начинает поиск вдоль направления неизмененного градиента.

Алгоритм 8.9. Метод сопряженных градиентов

Require: начальные значения параметров $\boldsymbol{\theta}_0$.
Require: обучающий набор m примеров
Инициализировать $\boldsymbol{\rho}_0 = 0$
Инициализировать $g_0 = 0$
Инициализировать $t = 1$
while критерий остановки не выполнен **do**
Инициализировать градиент $\boldsymbol{g}_t = 0$
Вычислить градиент: $\boldsymbol{g}_t \leftarrow (1/m)\nabla_{\boldsymbol{\theta}} \Sigma_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Вычислить $\beta_t = ((\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t)/\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}$ (метод Полака–Рибьера)
(Нелинейный метод сопряженных градиентов: иногда сбрасывать β_t в 0, например если t кратно некоторой константе k, скажем $k = 5$)
Вычислить направление поиска: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$
Произвести поиск с целью нахождения: $\varepsilon^* = \text{argmin}_\varepsilon (1/m)\Sigma_{i=1}^m L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \varepsilon\boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$

(Если функция стоимости строго квадратичная, не искать ε^*, а вычислить аналитически)
Применить обновление: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \varepsilon^* \boldsymbol{\rho}_t$
$t \leftarrow t + 1$
end while

Есть сообщения, что нелинейный метод сопряженных градиентов дает неплохие результаты при обучении нейронных сетей, хотя часто имеет смысл инициализировать оптимизацию, выполнив несколько итераций стохастического градиентного спуска, и только потом переходить к нелинейным сопряженным градиентам. Кроме того, хотя нелинейный алгоритм сопряженных градиентов традиционно считался пакетным методом, его мини-пакетные варианты успешно применялись для обучения нейронных сетей (Le et al., 2011). Позже были предложены адаптации метода сопряженных градиентов, например алгоритм масштабированных сопряженных градиентов (Moller, 1993).

8.6.3. Алгоритм BFGS

Алгоритм Бройдена–Флетчера–Гольдфарба–Шанно (BFGS) – попытка взять некоторые преимущества метода Ньютона без обременительных вычислений. В этом смысле BFGS аналогичен методу сопряженных градиентов. Однако в BFGS подход к аппроксимации обновления Ньютона более прямолинейный. Напомним, что обновление Ньютона определяется формулой

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \tag{8.32}$$

где $\boldsymbol{H}$ – гессиан J относительно $\boldsymbol{\theta}$, вычисленный в точке $\boldsymbol{\theta}_0$. Основная вычислительная трудность при применении обновления Ньютона – вычисление обратного гессиана $\boldsymbol{H}^{-1}$. В квазиньютоновских методах (из которых алгоритм BFGS самый известный) обратный гессиан аппроксимируется матрицей $\boldsymbol{M}_t$, которая итеративно уточняется в ходе обновлений низкого ранга.

Определение и вывод аппроксимации BFGS приводятся во многих учебниках по оптимизации, в т. ч. Luenberger (1984).

После нахождения аппроксимации гессиана $\boldsymbol{M}_t$ направление спуска $\boldsymbol{\rho}_t$ определяется по формуле $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$. В этом направлении производится линейный поиск для определения величины шага ε^*. Окончательное обновление параметров производится по формуле

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \varepsilon^* \boldsymbol{\rho}_t. \tag{8.33}$$

Как и в методе сопряженных градиентов, в алгоритме BFGS производится последовательность линейных поисков в направлениях, вычисляемых с учетом информации второго порядка. Но, в отличие от метода сопряженных градиентов, успех не так сильно зависит от того, находит ли линейный поиск точку, очень близкую к истинному минимуму вдоль данного направления. Поэтому BFGS тратит меньше времени на уточнение результатов каждого линейного поиска. С другой стороны, BFGS должен хранить обратный гессиан $\boldsymbol{M}$, для чего требуется память объема $O(n^2)$, поэтому BFGS непригоден для современных моделей глубокого обучения, насчитывающих миллионы параметров.

BFGS в ограниченной памяти (L-BFGS). Потребление памяти в алгоритме BFGS можно значительно уменьшить, если не хранить полную аппроксимацию обратного

гессиана $\boldsymbol{M}$. В алгоритме L-BFGS аппроксимация $\boldsymbol{M}$ вычисляется так же, как в BFGS, но, вместо того чтобы сохранять аппроксимацию между итерациями, делается предположение, что $\boldsymbol{M}^{(t-1)}$ – единичная матрица. При использовании совместно с точным линейным поиском направления, вычисляемые алгоритмом L-BFGS, являются взаимно сопряженными. Однако, в отличие от метода сопряженных градиентов, эта процедура ведет себя хорошо, даже когда линейный поиск находит только приближенный минимум. Описанную стратегию L-BFGS без запоминания можно обобщить, включив больше информации о гессиане; для этого нужно хранить некоторые векторы, используемые для обновления $\boldsymbol{M}$ на каждом шаге, тогда потребуется только память объемом $O(n)$.

8.7. Стратегии оптимизации и метаалгоритмы

Многие методы оптимизации – не совсем алгоритмы, а скорее общие шаблоны, которые можно специализировать и получить алгоритмы или подпрограммы, включаемые в различные алгоритмы.

8.7.1. Пакетная нормировка

Пакетная нормировка (Ioffe and Szegedy, 2015) – одна из наиболее интересных новаций в области оптимизации глубоких нейронных сетей – вообще алгоритмом не является. Это метод адаптивной перепараметризации, появившийся из-за трудностей обучения очень глубоких моделей.

Для очень глубоких моделей характерна композиция нескольких функций, или слоев. Градиент говорит, как обновлять каждый параметр в предположении, что другие слои не изменяются. На практике мы обновляем все слои одновременно. При выполнении обновления могут произойти неожиданности, потому что ко всем образующим композицию функции одновременно применяются обновления, вычисленные в предположении, что прочие функции сохраняют постоянство. Рассмотрим простой пример: предположим, что имеется глубокая нейронная сеть, в каждом слое которой находится по одному блоку и в скрытых слоях не используется функция активации: $\hat{y} = xw_1w_2w_3...w_l$. Здесь w_i – вес в i-м слое. Выход i-го слоя $h_i = h_{i-1}w_i$. Выход $\hat{y}$ линейно зависит от входа x, но нелинейно от весов w_i. Предположим, что наша функция стоимости дала градиент 1 по $\hat{y}$, поэтому мы хотим немного уменьшить $\hat{y}$. Тогда алгоритм обратного распространения может вычислить градиент $\boldsymbol{g} = \nabla_{\boldsymbol{w}}\hat{y}$. Посмотрим, что произойдет, когда мы произведем обновление $\boldsymbol{w} \leftarrow \boldsymbol{w} - \varepsilon\boldsymbol{g}$. Разложение $\hat{y}$ в ряд Тейлора до членов первого порядка предсказывает, что значение $\hat{y}$ уменьшится на $\varepsilon\boldsymbol{g}^\top\boldsymbol{g}$. Если бы мы хотели уменьшить $\hat{y}$ на 0.1, то, исходя из этой информации первого порядка, содержащейся в градиенте, могли бы установить скорость обучения ε равной $0.1/(\boldsymbol{g}^\top\boldsymbol{g})$. Однако фактическое обновление будет включать также эффекты второго, третьего, ..., l-го порядка. Новое значение $\hat{y}$ равно:

$$x(w_1 - \varepsilon g_1)(w_2 - \varepsilon g_2)...(w_l - \varepsilon g_l). \tag{8.34}$$

Пример члена второго порядка, возникающего при таком обновлении, – $\varepsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$. Этот член может быть пренебрежимо мал, если произведение $\prod_{i=3}^{l} w_i$ мало, а может быть экспоненциально велик, если веса в слоях с 3 по l больше 1. Поэтому очень трудно выбрать правильную скорость обучения, т. к. эффект обновления параметров одного слоя сильно зависит от всех остальных слоев. Алгоритмы оптимизации второго по-

рядка решают эту проблему, учитывая при вычислении обновления взаимодействия второго порядка, но, как мы видим, в очень глубоких сетях даже взаимодействия более высокого порядка могут быть существенными. Но уж если оптимизация второго порядка обходится дорого и обычно требует многочисленных аппроксимаций, не позволяющих точно учесть значимых взаимодействий второго порядка, то попытка построения алгоритма оптимизации n-го порядка при $n > 2$ выглядит совершенно безнадежной. Что же предпринять вместо этого?

Пакетная нормировка предлагает элегантный способ перепараметризации почти любой глубокой сети. Перепараметризация значительно снижает остроту проблемы координации обновлений между многими слоями. Пакетную нормировку можно применить к входному и любому скрытому слою сети. Пусть $\boldsymbol{H}$ – мини-пакет активаций нормируемого слоя, представленный в виде матрицы плана, так что активации для каждого примера занимают одну строку матрицы. Для нормировки заменим $\boldsymbol{H}$ матрицей

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{8.35}$$

где $\boldsymbol{\mu}$ – вектор средних всех блоков, а $\boldsymbol{\sigma}$ – вектор стандартных отклонений всех блоков. Приведенная выше формула выражает применение векторов $\boldsymbol{\mu}$ и $\boldsymbol{\sigma}$ к каждой строке матрицы $\boldsymbol{H}$. Внутри каждой строки операция применяется поэлементно, т. е. для нормировки $H_{i,j}$ нужно вычесть μ_j и разделить на σ_j. Остальная сеть работает с $\boldsymbol{H}'$ точно так же, как исходная работала с $\boldsymbol{H}$.

На этапе обучения

$$\boldsymbol{\mu} = \frac{1}{m}\sum_i \boldsymbol{H}_{i,:} \tag{8.36}$$

и

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m}\sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2}, \tag{8.37}$$

где δ – небольшое положительное число, например 10^{-8}, введенное, чтобы избежать неопределенного градиента $\sqrt{z}$ в точке $z = 0$. И важнейший момент – *мы выполняем обратное распространение сквозь эти операции*, чтобы вычислить среднее и стандартное отклонения и применить их к нормировке $\boldsymbol{H}$. Это означает, что градиент никогда не предлагает операцию, действие которой сводится просто к увеличению стандартного отклонения или среднего h_i; операции нормировки устраняют результат такого действия и обнуляют его компоненту в градиенте. Это и было главным нововведением пакетной нормировки. В прежних подходах к функции стоимости прибавлялись штрафы, направленные на то, чтобы блоки имели нормированные статистики активации, или после каждого шага градиентного спуска производилось вмешательство с целью перенормировать статистики блока. Первый подход обычно приводил к неидеальной нормировке, а последний – к значительным затратам времени впустую, потому что алгоритм обучения раз за разом предлагает изменить среднее и дисперсию, а на шаге нормировки это изменение отменяется. Пакетная нормировка перепараметризует модель, так что некоторые блоки всегда стандартизованы по определению, и тем самым ловко обходит обе проблемы.

На этапе тестирования $\boldsymbol{\mu}$ и $\boldsymbol{\sigma}$ можно заменить скользящими средними, подготовленными на этапе обучения. Это позволяет вычислять модель для одного примера, не прибегая к определениям $\boldsymbol{\mu}$ и $\boldsymbol{\sigma}$, которые зависят от всего мини-пакета. Возвращаясь к примеру $\hat{y} = xw_1w_2 \ldots w_l$, мы видим, что трудности обучения модели можно по большей части разрешить путем нормировки h_{l-1}. Предположим, что x имеет нормальное распределение с нулевым средним и единичной дисперсией. Тогда h_{l-1} тоже имеет нормальное распределение, потому что преобразование x в h_l линейно. Однако у h_{l-1} среднее уже не равно 0, а дисперсия не равна 1. После применения пакетной нормировки мы получаем $\hat{h}_{l-1}$ с восстановленными свойствами нулевого среднего и единичной дисперсии. Почти для любого обновления нижних слоев $\hat{h}_{l-1}$ сохраняет эти свойства. Тогда выход $\hat{y}$ можно обучить как простую линейную функцию $\hat{y} = w_l\hat{h}_{l-1}$. Обучение этой модели стало совсем простым делом, потому что параметры нижних слоев в большинстве случаев ни на что не влияют; их выход всегда перенормируется в нормальное распределение с нулевым средним и единичной дисперсией. В некоторых патологических ситуациях нижние слои могут оказывать влияние. Изменение одного из весов нижнего слоя на 0 может привести к вырожденному выходу, а изменение знака одного веса может обратить связь между $\hat{h}_{l-1}$ и y. Такие ситуации очень редки. Без нормировки почти любое обновление очень сильно влияло бы на статистику h_{l-1}. Таким образом, в результате пакетной нормировки обучить эту модель стало гораздо проще. Конечно, в этом примере ценой за простоту обучения стала бесполезность нижних уровней. В этой линейной ситуации нижние уровни не дают вредного эффекта, но и полезного тоже. Объясняется это тем, что в результате нормировки мы устранили статистики первого и второго порядка, а это всё, на что может повлиять линейная сеть. В глубокой нейронной сети с нелинейными функциями активации нижние уровни могут выполнять нелинейные преобразования данных, поэтому остаются полезными. Действие пакетной нормировки направлено на стандартизацию только среднего и дисперсии каждого блока с целью стабилизировать обучение, но она не препятствует изменению связей между блоками и нелинейных статистик одного блока.

Поскольку последний слой сети способен обучиться линейному преобразованию, мы на самом деле можем попробовать удалить все линейные связи между блоками в пределах одного слоя. Именно так и поступили авторы работы Desjardins et al. (2015), которая подсказала идею пакетной нормировки. К сожалению, исключение всех линейных взаимодействий обходится гораздо дороже стандартизации среднего и стандартного отклонений каждого отдельного блока, так что пакетная нормировка до сих пор остается наиболее практичным решением.

Нормировка среднего и стандартного отклонений блока может снизить выразительную мощность нейронной сети, содержащей этот блок. Для сохранения выразительной мощности обычно заменяют пакет активаций скрытых блоков $\boldsymbol{H}$ на $\gamma\boldsymbol{H}' + \beta$, а не просто на нормированную матрицу $\boldsymbol{H}'$. Переменные γ и β – обученные параметры, благодаря которым новая величина может иметь произвольные среднее и стандартное отклонения. На первый взгляд, это кажется бессмысленным – зачем было устанавливать среднее в 0, а потом вводить параметр, который позволяет снова переустановить его в произвольное значение β? Да затем, что новая параметризация может представить то же самое семейство функций от входных данных, что и старая, но при этом обладает другой динамикой обучения. В старой параметризации среднее $\boldsymbol{H}$ определялось сложным взаимодействием между параметрами на уровнях ниже $\boldsymbol{H}$.

В новой же параметризации среднее $\gamma \boldsymbol{H}' + \beta$ определяется только величиной β. При новой параметризации модель гораздо легче обучить методом градиентного спуска.

Большинство слоев нейронных сетей имеет вид $\phi(\boldsymbol{XW} + \boldsymbol{b})$, где ϕ – фиксированная нелинейная функция активации, например, преобразование линейной ректификации. Естественно спросить, следует ли применять пакетную нормировку ко входу $\boldsymbol{X}$ или к уже преобразованному значению $\boldsymbol{XW} + \boldsymbol{b}$. В работе Ioffe and Szegedy (2015) рекомендуется последнее. Точнее говоря, $\boldsymbol{XW} + \boldsymbol{b}$ следует заменить результатом нормировки $\boldsymbol{XW}$. Член смещения нужно опустить, потому что он становится избыточным ввиду параметра β, применяемого в ходе перепараметризации. Входом слоя обычно является выход нелинейной функции активации (например, функции линейной ректификации) предыдущего слоя. Поэтому статистика входа сильнее отличается от нормального распределения и хуже поддается стандартизации посредством линейных операций. В сверточных сетях (см. главу 9) важно применять одну и ту же нормировку μ и σ в каждой точке пространства в карте признаков, чтобы статистика карты признаков оставалась одинаковой вне зависимости от положения в пространстве.

8.7.2. Покоординатный спуск

В некоторых случаях задачу оптимизации можно быстро решить, разбив на отдельные части. Если минимизировать $f(\boldsymbol{x})$ сначала по переменной x_i, затем по переменной x_j и т. д., перебрав в цикле все переменные, то мы гарантированно окажемся в (локальном) минимуме. Такой подход называется **покоординатным спуском**, поскольку в каждый момент времени производится оптимизация по одной координате. В более общем случае **блочно-покоординатного спуска** одновременно производится минимизация по подмножеству переменных. Термин «покоординатный спуск» часто употребляется не только в своем строгом смысле, но и для обозначения блочно-покоординатного спуска.

Покоординатный спуск наиболее осмыслен, когда различные переменные в задаче оптимизации можно разбить на группы с относительно изолированными ролями или когда оптимизация по одной группе переменных значительно эффективнее, чем по всем. Рассмотрим, к примеру, такую функцию стоимости:

$$J(\boldsymbol{H},\boldsymbol{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (\boldsymbol{X} - \boldsymbol{W}^\top \boldsymbol{H})^2_{i,j}. \tag{8.38}$$

Эта функция описывает проблему обучения, которая называется разреженным кодированием. Цель состоит в том, чтобы найти матрицу весов $\boldsymbol{W}$, которая может линейно декодировать матрицу значений активации $\boldsymbol{H}$ и реконструировать обучающий набор $\boldsymbol{X}$. В большинстве применений разреженного кодирования участвует также снижение весов или ограничение на нормы столбцов $\boldsymbol{W}$, чтобы предотвратить патологические решения с очень малой $\boldsymbol{H}$ и большой $\boldsymbol{W}$.

Функция J невыпуклая. Однако мы можем разбить входы алгоритма обучения на два множества: словарные параметры $\boldsymbol{W}$ и представление кода $\boldsymbol{H}$. Минимизация целевой функции по любому из этих двух множеств – выпуклая задача. Поэтому метод блочно-покоординатного спуска позволяет использовать эффективные алгоритмы выпуклой оптимизации, выполнив сначала оптимизацию по $\boldsymbol{W}$ с фиксированным $\boldsymbol{H}$, а затем оптимизацию по $\boldsymbol{H}$ с фиксированным $\boldsymbol{W}$.

Покоординатный спуск – не самая удачная стратегия, когда значение одной переменной сильно влияет на оптимальное значение другой, как в функции $f(\boldsymbol{x}) =$

$= (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$, где α – положительная постоянная. Для минимизации первого члена нужно, чтобы переменные мало отличались друг от друга, а для минимизации второго – чтобы обе были близки к нулю. Минимум достигается, когда обе переменные равны 0. Методом Ньютона эту задачу можно было бы решить за один шаг, потому что она квадратичная и положительно определенная. Однако при малых α метод покоординатного спуска сходится очень медленно, потому что первый член не дает изменить одну переменную, так чтобы ее значение сильно отличалось от значения второй переменной.

8.7.3. Усреднение Поляка

Усреднение Поляка (Polyak and Juditsky, 1992) заключается в усреднении нескольких точек на траектории алгоритма оптимизации в пространстве параметров. Если на протяжении t итераций градиентного спуска алгоритм посетил точки $\boldsymbol{\theta}^{(1)}, ..., \boldsymbol{\theta}^{(t)}$, то выходом алгоритма усреднения Поляка будет $\hat{\boldsymbol{\theta}}(t) = (1/t)\Sigma_i \boldsymbol{\theta}^{(i)}$. Для некоторых классов задач, например градиентного спуска в применении к выпуклым задачам, этот подход дает сильные гарантии сходимости. В применении к нейронным сетям обоснование скорее эвристическое, но на практике дает неплохие результаты. Основная идея состоит в том, что алгоритм оптимизации может несколько раз пересечь овраг, так и не попав в точку на его дне. Но усреднение по точкам на каждом склоне оврага должно быть расположено ближе ко дну.

В невыпуклых задачах траектория оптимизации может оказаться очень сложной и заходить во много различных областей. Вряд ли полезно включать точки, посещенные в далеком прошлом, которые могут быть отделены от текущей точки большими барьерами на поверхности функции стоимости. Поэтому, когда усреднение Поляка применяется к невыпуклым задачам, обычно берут экспоненциально затухающее скользящее среднее:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha\hat{\boldsymbol{\theta}}^{(t-1)} + (1 - \alpha)\boldsymbol{\theta}^{(t)}. \tag{8.39}$$

Такой подход используется в многочисленных приложениях. Недавний пример см. в работе Szegedy et al. (2015).

8.7.4. Предобучение с учителем

Иногда прямое обучение модели для решения конкретной задачи – слишком амбициозная цель, если модель сложная и с трудом поддается оптимизации или сама задача очень трудна. Бывает, что эффективнее обучить более простую модель для решения той же задачи, а затем усложнить ее. Или обучить модель решению более простой задачи, а затем перейти к настоящей задаче. Стратегии обучения простых моделей на простых задачах, перед тем как приступить к обучению желаемой модели на желаемой задаче, имеют собирательное название: **предобучение**.

Жадные алгоритмы разбивают задачу на несколько компонент, а затем находят оптимальное решение для каждой компоненты в отдельности. К сожалению, не гарантируется, что комбинация оптимальных компонент дает оптимальное решение задачи в целом. Тем не менее жадные алгоритмы вычислительно могут оказаться гораздо дешевле алгоритмов, ищущих наилучшее совместное решение, а их качество, хоть и неоптимальное, зачастую приемлемо. После завершения жадного алгоритма можно выполнить фазу **окончательной настройки**, когда алгоритм совместной опти-

мизации ищет оптимальное решение задачи в целом. Если инициализировать алгоритм совместной оптимизации решением, найденным жадным алгоритмом, то можно существенно ускорить поиск окончательного решения и улучшить его качество.

Предобучение, и особенно жадное предобучение, встречается в глубоком обучении повсеместно. В этом разделе мы опишем алгоритмы предобучения, в которых задача обучения с учителем разбивается на несколько более простых задач такого же типа. Этот подход называется **жадным предобучением с учителем**.

В оригинальном варианте жадного предобучения с учителем (Bengio et al., 2007) каждый этап состоит из задачи обучения с учителем, содержащей только часть слоев финальной нейронной сети. На рис. 8.7 показан пример жадного предобучения с учителем, когда каждый скрытый слой предварительно обучен в составе мелкого МСП с учителем, принимающего на входе выход предыдущего скрытого слоя. Вместо предобучения по одному слою за раз в работе Simonyan and Zisserman (2015) производится предобучение глубокой сверточной сети (11 слоев весов), а затем первые четыре и последние три слоя этой сети используются для инициализации еще более глубоких сетей (до 19 слоев). Средние слои новой, очень глубокой сети инициализируются случайным образом. Затем производится совместное обучение новой сети. Другой вариант, исследованный в работе Yu et al. (2010), – применять *выходы* ранее обученных МСП, а также первоначальные входные данные в качестве входов каждого дополнительного этапа.

Почему жадное предобучение с учителем дает какой-то эффект? В пионерской работе Bengio et al. (2007) высказана гипотеза, что оно дает более правильное направление промежуточным слоям глубокой иерархии. В общем случае предобучение может оказаться полезным как для оптимизации, так и для обобщаемости.

Обобщением предобучения с учителем является перенос обучения (transfer learning): в работе Yosinski et al. (2014) была предобучена глубокая сверточная сеть с восьмью уровнями весов на наборе задач (подмножество, состоящее из 1000 категорий объектов из набора ImageNet), а затем сеть того же размера была инициализирована первыми k слоями первой сети. Далее все слои второй сети (верхние слои которой были инициализированы случайным образом) были совместно обучены для выполнения различных наборов задач (другое подмножество ImageNet из 1000 категорий объектов), но число обучающих примеров было меньше, чем для первого набора. Другие подходы к переносу обучения в контексте нейронных сетей обсуждаются в разделе 15.2.

Сюда же можно отнести подход FitNets (Romero et al., 2015). Он начинается с обучения сети достаточно малой глубины и достаточно большой ширины (число блоков в слое), чтобы ее было легко обучить. Затем эта сеть становится **учителем** для второй сети, называемой **учеником**. Сеть-ученик гораздо глубже и уже (от 11 до 19 слоев), и обучить ее методом СГС при нормальных обстоятельствах было бы затруднительно. Обучение сети-ученика облегчается тем, что ученик учится предсказывать не только выход исходной задачи, но и значение среднего слоя сети-учителя. Эта дополнительная задача предоставляет ряд рекомендаций по использованию скрытых слоев и может упростить задачу оптимизации. Вводятся дополнительные параметры, чтобы восстановить средний слой пятислойной сети-учителя по среднему слою более глубокой сети-ученика. Но целью является не предсказание финального класса, а предсказание скрытого среднего слоя сети-учителя. Таким образом, у нижних слоев сети-ученика две цели: помочь выходам выполнить свою задачу, а также предсказать

промежуточный слой сети-учителя. Хотя обучать узкую и глубокую сеть труднее, чем широкую и мелкую, но узкая и глубокая сеть лучше обобщается и, безусловно, для узкой сети с небольшим числом параметров вычислительная стоимость обучения ниже. Без рекомендаций по скрытому слою сеть-ученик в экспериментах вела себя очень плохо – как на обучающем, так и на тестовом наборе. Таким образом, рекомендации по средним слоям, быть может, являются одним из средств обучения нейронных сетей, с трудом поддающихся обучению другими методами. Но не исключено, что другие методы оптимизации или изменение архитектуры тоже могло бы решить проблему.

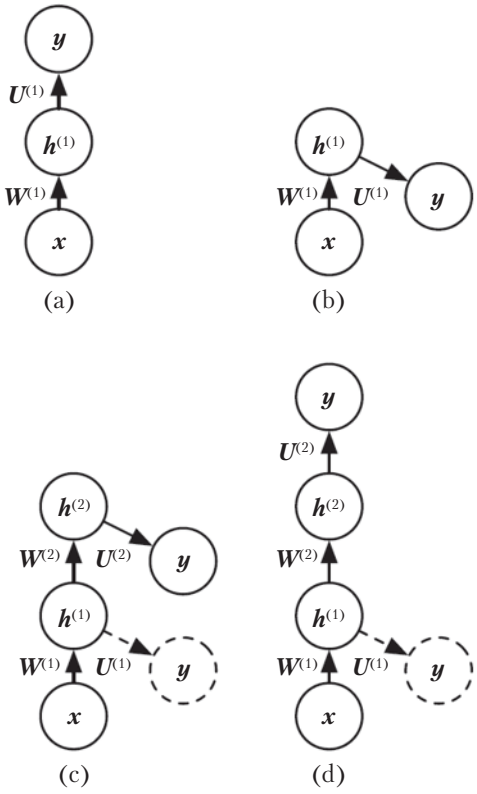

Рис. 8.7 ❖ Иллюстрация одного из видов жадного предобучения с учителем (Bengio et al., 2007). (*a*) Начинаем с обучения достаточно мелкой архитектуры. (*b*) Другой вид той же архитектуры. (*c*) Оставляем только сигналы между входным и скрытым слоями исходной сети и отбрасываем сигналы между скрытым и выходным слоями. Передаем выход первого скрытого слоя на вход еще одного однослойного МСП с учителем, который обучается с той же целевой функцией, что и первая сеть, и таким образом добавляем второй скрытый слой. Эту операцию можно повторить сколько угодно раз. (*d*) Тот же результат, представленный в виде сети прямого распространения. Чтобы улучшить оптимизацию, мы можем произвести совместную окончательную настройку всех слоев – либо в самом конце, либо на каждом этапе процесса

8.7.5. Проектирование моделей с учетом простоты оптимизации

Лучшая стратегия улучшения оптимизации не всегда связана с улучшением алгоритма. Вместо этого можно изначально проектировать модели так, чтобы их было проще оптимизировать.

В принципе, можно было бы использовать немонотонные функции активации, которые то возрастают, то убывают, но тогда оптимизация оказалась бы крайне трудным делом. На практике *важнее выбирать семейство моделей, легко поддающееся оптимизации, чем мощный алгоритм оптимизации*. Прогресс в области нейронных сетей за последние тридцать лет в основном был связан именно с изменением семейства моделей. Стохастический градиентный спуск с учетом импульса, использовавшийся для обучения сетей еще в 1980-х годах, и по сей день применяется в самых передовых приложениях.

Говоря конкретно, в современных нейронных сетях *предпочтение отдается* линейным преобразованиям между слоями и функциями активации, дифференцируемым почти всюду и имеющим значительный наклон на больших участках своей области определения. Такие новации в области проектирования моделей, как LSTM, блоки линейной ректификации и maxout-блоки, все направлены на использование более линейных функций, чем в предыдущих моделях типа глубоких сетей с сигмоидными блоками. У таких моделей есть полезные свойства, упрощающие оптимизацию. Градиент распространяется сквозь много слоев, при условии что у якобиана линейного преобразования имеются разумные сингулярные значения. К тому же линейные функции монотонно возрастают в одном направлении, так что даже если выход модели очень далек от правильного, из вычисления градиента сразу становится ясно, в каком направлении должен сместиться выход, чтобы уменьшить функцию потерь. Иными словами, современные нейронные сети проектируются так, чтобы *локальная* информация о градиенте достаточно хорошо соответствовала движению в сторону далеко находящегося решения.

Есть и другие стратегии проектирования моделей, способствующие упрощению оптимизации. Например, линейные пути или прямые связи между слоями уменьшают длину кратчайшего пути от параметров нижних слоев к выходу, а потому смягчают проблему исчезающего градиента (Srivastava et al., 2015). К идее прямых связей близка идея о добавлении дополнительных копий выходов, которые были бы соединены с промежуточными слоями сети, как в GoogLeNet (Szegedy et al., 2014в а) и сетях с глубоким проникновением учителя (deeply supervised nets) (Lee et al., 2014). Эти «вспомогательные головы» обучаются решать ту же задачу, что основной верхний слой сети, чтобы нижние слои получали больший градиент. По завершении обучения вспомогательные головы можно отбросить. Это альтернатива стратегиям предобучения, описанным в предыдущем разделе. При таком подходе можно совместно обучать все слои на одной стадии, но изменить архитектуру, так чтобы промежуточные слои (особенно ниже расположенные) могли получать рекомендации о том, что делать, по более короткому пути. Эти рекомендации несут нижним слоям сигнал об ошибке.

8.7.6. Методы продолжения и обучение по плану

Как было сказано в разделе 8.2.7, многие проблемы оптимизации проистекают из глобальной структуры функции стоимости, их невозможно разрешить просто за счет улучшения оценки направления локального обновления. Основная стратегия преодоления таких проблем – попытаться присвоить параметрам начальные значения,

находящиеся в области, связанной с решением коротким путем в пространстве параметров, который можно найти методом локального спуска.

Методы продолжения – это семейство стратегий, которые упрощают оптимизацию посредством выбора таких начальных точек, чтобы локальная оптимизация проходила в основном в областях пространства с хорошим поведением. Идея заключается в том, чтобы построить последовательность целевых функций от одних и тех же параметров. Чтобы минимизировать функцию стоимости $J(\boldsymbol{\theta})$, мы строим новые функции стоимости $\{J^{(0)}, ..., J^{(n)}\}$. Сложность этих функций постепенно нарастает, так что минимизировать $J^{(0)}$ сравнительно легко, а функция $J^{(n)}$, которую труднее всего минимизировать, совпадает с $J(\boldsymbol{\theta})$ – истинной функцией стоимости, ради которой все и затевалось. Говоря, что $J^{(i)}$ проще $J^{(i+1)}$, мы имеем в виду, что она хорошо себя ведет в большей части пространства $\boldsymbol{\theta}$. Случайно выбранные начальные значения с большей вероятностью окажутся в области, где метод локального спуска способен минимизировать функцию стоимости, просто потому что эта область больше. Последовательность функций стоимости проектируется так, чтобы решение одной функции было хорошим начальным приближением для другой. Таким образом, мы сначала находим решение легкой задачи, а затем постепенно уточняем его для решения все более трудных задач, пока не найдем решения исходной задачи.

Традиционные методы продолжения (которые появились задолго до применения этой идеи к обучению нейронных сетей) обычно основываются на сглаживании целевой функции. В работе Wu (1997) приведены пример такого метода и обзор родственных ему. Методы продолжения также тесно связаны с имитацией отжига, когда к параметрам добавляется шум (Kirkpatrick et al., 1983). В последние годы методы продолжения применялись чрезвычайно успешно. В статье Mobahi and Fisher (2015) имеется обзор недавних работ, особенно в области приложений ИИ.

Традиционно методы продолжения проектировались прежде всего для преодоления проблем, связанных с наличием локальных минимумов. Точнее, их цель – найти глобальный минимум, несмотря на присутствие многих локальных. Для этого строятся более простые функции стоимости путем «размытия» исходной. Эту операцию размытия можно выполнить, если выборочно аппроксимировать функцию

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}'). \tag{8.40}$$

Интуитивное соображение в пользу этого подхода заключается в том, что некоторые невыпуклые функции становятся приближенно выпуклыми после размытия. Во многих случаях размытие сохраняет достаточно информации о положении глобального минимума, так что его можно найти, решая варианты задачи с уменьшающейся степенью размытия. Но эта идея может не сработать по трем причинам. Прежде всего нам, возможно, и удастся определить последовательность функций стоимости, в которой первая функция будет выпуклой, и, проследив оптимум при переходе от одной функции к другой, в итоге найти глобальный минимум, но количество промежуточных функций будет настолько велико, что стоимость всей процедуры останется слишком большой. NP-трудная задача остается NP-трудной, даже если методы продолжения к ней применимы. Другие две причины неудачи соответствуют неприменимости метода в принципе. Во-первых, не исключено, что функция не становится выпуклой ни при какой степени размытия. Взять, к примеру, функцию $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$. Во-вторых, функция может оказаться выпуклой после размытия, но прослеживание ее минимума приводит к локальному, а не глобальному минимуму исходной функции стоимости.

Хотя методы продолжения изначально разрабатывались для решения проблемы локальных минимумов, в контексте оптимизации нейронных сетей эта проблема уже не считается самой животрепещущей. Тем не менее методы продолжения попрежнему полезны. Упрощение целевой функции может устранить плоские участки, уменьшить дисперсию оценки градиента, улучшить обусловленность матрицы Гессе или сделать еще что-то, что либо облегчит вычисление локальных обновлений, либо улучшит соответствие между локальными обновлениями направлений и движением в сторону глобального решения.

В работе Bengio et al. (2009) отмечено, что подход, называемый **обучением по плану** (curriculum learning), или **шейпингом** (shaping), можно интерпретировать как метод продолжения. В основе обучения по плану лежит идея планирования процесса обучения, когда начинают с простых понятий и постепенно вводят более сложные. Ранее эта базовая стратегия применялась, чтобы ускорить обучение животных (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009), и в машинном обучении (Solomonoff, 1989; Elman, 1993; Sanger, 1994). В работе Bengio et al. (2009) приведено ее обоснование как метода продолжения, в котором простота предшествующих функций $J^{(i)}$ обеспечивается увеличением влияния более простых примеров (либо за счет того, что их вкладу в функцию стоимости назначаются бóльшие коэффициенты, либо потому что они выбираются чаще). Экспериментально продемонстрировано, что при решении крупномасштабной задачи моделирования языка нейронной сетью обучение по плану улучшает результаты. Обучение по плану успешно применялось к широкому кругу задач в области обработки естественных языков (Spitkovsky et al., 2010; Collobert et al., 2011a; Mikolov et al., 2011b; Tu and Honavar, 2011) и компьютерного зрения (Kumar et al., 2010; Lee and Grauman, 2011; Supancic and Ramanan, 2013). Также установлено, что обучение по плану согласуется с тем, как *преподает* человек (Khan et al., 2011): преподаватель сначала показывает более простые и прототипичные примеры, а затем помогает обучаемому уточнить поверхность решений на менее очевидных случаях. Такие стратегии не только *более эффективны* для обучения людей, чем основанные на равномерной выборке примеров, но и могут повысить эффективность других стратегий обучения (Basu and Christensen, 2013).

Еще один важный вклад в исследования в области обучения по плану связан с обучением рекуррентных нейронных сетей улавливанию долговременных зависимостей. В работе Zaremba and Sutskever (2014) обнаружено, что гораздо лучшие результаты получаются при использовании *стохастического плана*, когда обучаемому всегда предъявляется случайная смесь простых и трудных примеров, но средняя доля трудных примеров (тех, в которых имеются долговременные зависимости) постепенно увеличивается. Когда использовался детерминированный план, никакого улучшения по сравнению с эталоном (обычное обучение на полном обучающем наборе) не наблюдалось.

Итак, мы описали базовое семейство моделей нейронных сетей и способы их регуляризации и оптимизации. В последующих главах мы займемся частными случаями этого семейства, когда сеть масштабируется на очень большие объемы данных и обрабатываются данные со специальной структурой. Рассмотренные выше методы оптимизации часто применимы к таким специализированным архитектурам после небольшой модификации или даже в неизменном виде.

Глава 9

Сверточные сети

Сверточная сеть (LeCun, 1989), она же **сверточная нейронная сеть** (CHC), – это специальный вид нейронной сети для обработки данных с сеточной топологией. Примерами могут служить временные ряды, которые можно рассматривать как одномерную сетку примеров, выбираемых через регулярные промежутки времени, а также изображения, рассматриваемые как двумерная сетка пикселей. Сверточные сети добились колоссального успеха в практических приложениях. Своим названием они обязаны использованию математической операции **свертки**. Свертка – это особый вид линейной операции. *Сверточные сети – это просто нейронные сети, в которых вместо общей операции умножения на матрицу, по крайней мере в одном слое, используется свертка.*

В этой главе мы сначала опишем, что такое свертка. Затем мы объясним причины использования свертки в нейронных сетях. Далее будет описана операция **пулинга**, применяемая почти во всех сверточных сетях. Обычно операция, используемая в сверточной нейронной сети, не вполне соответствует определению свертки в других областях, например в инженерных дисциплинах и в чистой математике. Мы опишем несколько вариантов функции свертки, широко применяемых в нейронных сетях. Мы также покажем, как можно применить свертку к различным видам данных в пространствах разной размерности. Затем мы обсудим, как повысить эффективность свертки. Сверточные сети – яркий пример того, как принципы нейробиологии оказывают влияние на глубокое обучение. Мы обсудим эти принципы и завершим главу замечаниями о роли сверточных сетей в истории глубокого обучения. В этой главе не затрагивается вопрос о выборе архитектуры сверточной сети. Цель главы – описать инструментарий, предоставляемый сверточными сетями, а в главе 11 мы приведем общие рекомендации по выбору инструментов в конкретных условиях. Прогресс в изучении сверточных сетей настолько быстрый, что сообщения о новой оптимальной архитектуре для данного эталонного теста появляются через каждые несколько недель или месяцев, поэтому называть какую-то архитектуру лучшей в книге пока преждевременно. Тем не менее все лучшие архитектуры скомпонованы из описанных в книге строительных блоков.

9.1. Операция свертки

В самом общем виде свертка – это операция над двумя функциями вещественного аргумента. Чтобы обосновать определение свертки, начнем с примеров возможных функций.

Допустим, что мы следим за положением космического корабля с помощью лазерного датчика. Наш датчик выдает единственное значение $x(t)$, положение корабля

в момент t. Переменные x и t принимают вещественные значения, т. е. показания датчика в любые два момента времени могут различаться.

Теперь предположим, что датчик подвержен помехам. Чтобы получить менее зашумленную оценку положения корабля, необходимо усреднить несколько результатов измерений. Разумеется, недавние измерения более важны, поэтому мы хотим вычислять взвешенное среднее, придавая недавним измерениям больший вес. Для этого можно воспользоваться весовой функцией $w(a)$, где a – давность измерения. Применив такую операцию усреднения в каждый момент времени, мы получим новую функцию, которая дает сглаженную оценку положения космического корабля:

$$s(t) = \int x(a)w(t-a)da. \tag{9.1}$$

Эта операция называется **сверткой** и обычно обозначается звездочкой:

$$s(t) = (x * w)(t). \tag{9.2}$$

В нашем примере w должна быть функцией плотности вероятности, иначе усреднения не получится. Кроме того, w должна быть равна 0 для всех отрицательных значений аргумента, иначе она будет способна заглядывать в будущее, что вряд ли в пределах наших возможностей. Но эти ограничения характерны только для данного примера. В общем случае свертку можно определить для любых функций, для которых определен показанный выше интеграл, и использовать не только для получения взвешенного среднего.

В терминологии сверточных сетей первый аргумент (в нашем примере функция x) называется **входом**, а второй (функция w) – **ядром**. Выход иногда называют **картой признаков**.

Лазерных датчиков, способных выдавать результаты измерений в любой момент времени, в действительности не бывает. Обычно при работе с данными в компьютере время дискретизировано, и наш датчик будет выдавать данные через регулярные интервалы. Пожалуй, было бы реалистичнее предположить, что лазер производит измерения раз в секунду. Индекс момента времени t может принимать только целые значения. Если теперь предположить, что x и w определены только для целых t, то мы получим определение дискретной свертки:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{9.3}$$

В приложениях машинного обучения входом обычно является многомерный массив данных, а ядром – многомерный массив параметров, адаптированных алгоритмом обучения. Будем называть эти массивы тензорами. Поскольку каждый элемент входа и ядра должен храниться отдельно в явном виде, обычно предполагается, что эти функции равны нулю всюду, кроме конечного множества точек, для которых мы храним значения. На практике это означает, что сумму от минус до плюс бесконечности можно заменить суммированием по конечному числу элементов массива.

Наконец, мы часто используем свертки сразу по нескольким осям. Например, если входом является двумерное изображение I, то и ядро должно быть двумерным:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{9.4}$$

Операция свертки коммутативна, поэтому формулу можно записать и так:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n). \tag{9.5}$$

Обычно вторую формулу проще реализовать в библиотеке машинного обучения, поскольку диапазон допустимых значений m и n меньше.

Свойство коммутативности свертки имеет место, потому что мы **отразили** ядро относительно входа, т. е. при увеличении m индекс входа увеличивается, а индекс ядра уменьшается. Единственная причина такого отражения – обеспечить коммутативность. И хотя коммутативность полезна для доказательства теорем, в реализации нейронных сетей она обычно роли не играет. Вместо этого во многих библиотеках реализована родственная функция – **перекрестная корреляция** – та же свертка, только без отражения ядра:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n). \tag{9.6}$$

Во многих библиотеках машинного обучения реализована именно перекрестная корреляция, но называется она сверткой. Далее мы будем придерживаться этого соглашения и называть сверткой обе операции, а в тех местах, где отражение ядра имеет значение, будем это явно оговаривать. В контексте машинного обучения алгоритм обучения ставит найденные значения ядра в правильную позицию, поэтому если алгоритм основан на свертке с отражением ядра, то он обучит ядро, отраженное относительно того, которое обучено алгоритмом со сверткой без отражения. Редко бывает так, чтобы свертка использовалась в машинном обучении сама по себе; чаще она комбинируется с другими функциями, и такие комбинации не коммутативны вне зависимости от того, используется в ядре-свертке отражение или нет.

На рис. 9.1 приведен пример свертки (без отражения ядра) в применении к двумерному тензору.

Дискретную свертку можно рассматривать как умножение на матрицу, на элементы которой наложены некоторые ограничения. Например, в случае одномерной дискретной свертки каждая строка матрицы должна быть равна предыдущей, сдвинутой на один элемент. Это утверждение называется **теоремой Теплица**. В двумерном случае свертке соответствует **дважды блочно-циркулянтная матрица**. Помимо ограничений на равенство некоторых элементов, свертке обычно соответствует сильно разреженная матрица (в которой большинство элементов равно нулю). Связано это с тем, что ядро, как правило, гораздо меньше входного изображения. Любой алгоритм нейронной сети, основанный на умножении матриц и не зависящий от особенностей структуры этих матриц, должен работать и со сверткой без каких-либо модификаций самой сети. В типичных сверточных нейронных сетях все же используются особенности структуры, чтобы организовать эффективную обработку больших входов, но с теоретической точки зрения это необязательно.

9.2. Мотивация

Со сверткой связаны три важные идеи, которые помогают улучшить систему машинного обучения: **разреженные взаимодействия**, **разделение параметров** и **эквивариантные представления**. Кроме того, свертка предоставляет средства для работы со входами переменного размера. Опишем эти идеи поочередно.

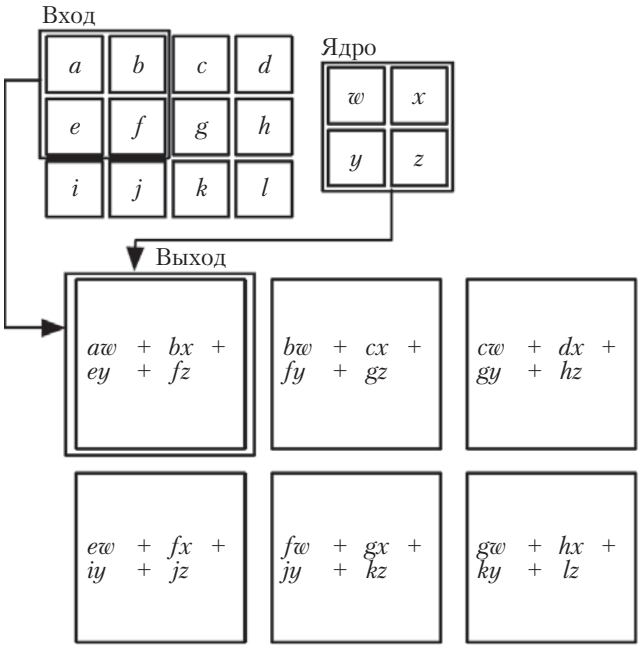

Рис. 9.1 ❖ Пример двумерной свертки без отражения ядра. Выход ограничен только теми позициями, для которых ядро целиком укладывается в изображение; в некоторых контекстах такая свертка называется «допустимой». Прямоугольник с указывающими на него стрелками показывает, как левый верхний элемент выходного тензора образуется путем применения ядра к соответствующей левой верхней области входного тензора

В слоях традиционной нейронной сети применяется умножение на матрицу параметров, в которой взаимодействие между каждым входным и каждым выходным блоками описывается отдельным параметром. Это означает, что каждый выходной блок взаимодействует с каждым входным блоком. Напротив, в сверточных сетях взаимодействия обычно разреженные (это свойство называют еще **разреженной связностью**, или разреженными весами). Достигается это за счет того, что ядро меньше входа. Например, входное изображение может содержать тысячи или миллионы пикселей, но небольшие значимые признаки, например границы, можно обнаружить с помощью ядра, охватывающего всего десятки или сотни пикселей. Следовательно, нужно хранить меньше параметров, а это снижает требования модели к объему памяти и повышает ее статистическую эффективность. Кроме того, для вычисления выхода потребуется меньше операций. Все вместе обычно намного повышает эффективность сети. Если имеется m входов и n выходов, то для умножения матриц нужно $m \times n$ параметров, и сложность практически используемых алгоритмов составляет $O(m \times n)$ (в расчете на один пример). Если ограничить число соединений с каждым выходом величиной k, то потребуется только $k \times n$ параметров, и сложность составит $O(k \times n)$. Во многих практических приложениях можно получить хорошее качество на задаче машинного обучения, когда k на несколько порядков меньше m. Графически разреженная связность иллюстрируется на рис. 9.2 и 9.3. В глубокой сверточной сети блоки нижних уровней могут косвенно взаимодействовать с большей частью сети,

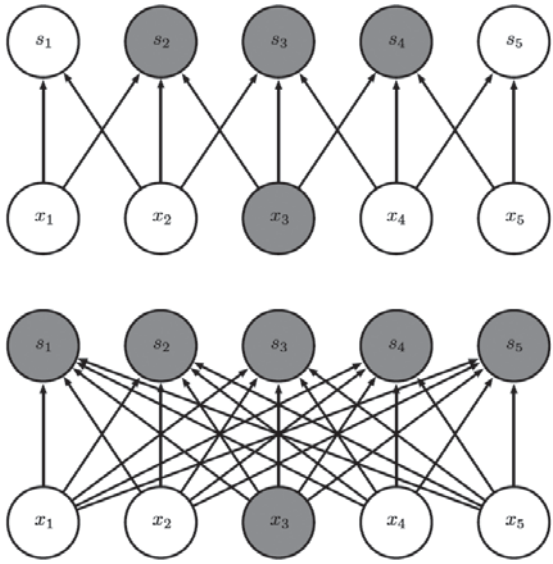

Рис. 9.2 ❖ Разреженная связность при взгляде снизу. Выделен один входной блок x_3 и выходные блоки в слое s, на которые этот блок влияет. (*Вверху*) Если s образован сверткой с ядром ширины 3, то x_3 влияет только на три выхода. (*Внизу*) Если s образован умножением на матрицу, то связность уже не разреженная, поэтому x_3 влияет на все выходы

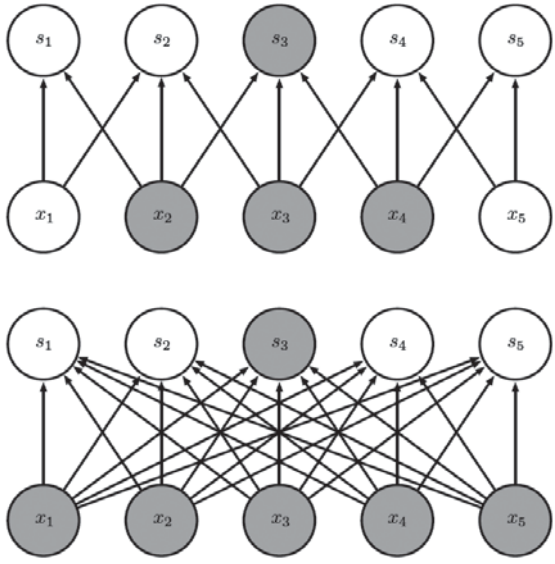

Рис. 9.3 ❖ Разреженная связность при взгляде сверху. Выделен один выходной блок s_3 и выходные блоки в слое x, которые на него влияют. Совокупность этих блоков называется рецептивным полем s_3. (*Вверху*) Если s образован сверткой с ядром ширины 3, то на s_3 влияют только три входа. (*Внизу*) Если s образован умножением на матрицу, то связность уже не разреженная, поэтому на s_3 влияют все входы

как показано на рис. 9.4. Это дает сети возможность эффективно описывать сложные взаимодействия между многими переменными путем составления из простых строительных блоков, каждый из которых описывает только разреженные взаимодействия.

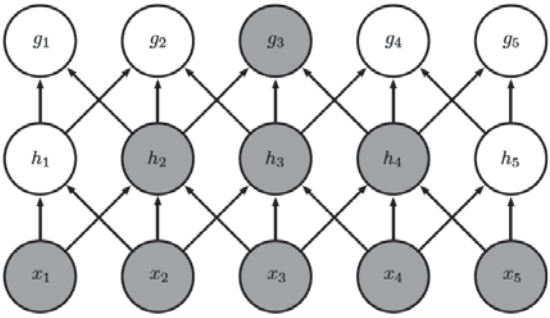

Рис. 9.4 ❖ Рецептивное поле блоков в глубоких слоях сверточной сети больше рецептивного поля в слоях, близких к поверхности. Этот эффект усиливается, если сеть включает такие архитектурные особенности, как свертка с шагом (рис. 9.12) или пулинг (раздел 9.3). Это означает, что хотя прямые связи в сверточной сети действительно очень разрежены, блоки в глубоких слоях могут быть косвенно связаны со всем входным изображением или с большей его частью

Под **разделением параметров** понимают, что один и тот же параметр используется в нескольких функциях модели. В традиционной нейронной сети каждый элемент матрицы весов используется ровно один раз при вычислении выхода слоя. Он умножает на один элемент входа, и больше мы к нему никогда не возвращаемся. Вместо употребления термина «разделение параметров» можно сказать, что в сети присутствуют **связанные веса**, поскольку значение веса, примененного к одному входу, связано со значением веса, примененного где-то еще. В сверточной нейронной сети каждый элемент ядра применяется к каждой позиции входа (за исключением, быть может, некоторых граничных пикселей – в зависимости от того, как решено обрабатывать границу). Разделение параметров означает, что вместо обучения отдельного набора параметров для каждой точки мы должны обучить только один набор. Это не влияет на время прямого распространения – оно по-прежнему имеет порядок $O(k \times n)$, – но дополнительно уменьшает требования к объему памяти: достаточно хранить k параметров. Напомним, что k обычно на несколько порядков меньше m. Поскольку величины m и n приблизительно равны, то k практически несущественно по сравнению с $m \times n$. Таким образом, свертка многократно эффективнее умножения матриц с точки зрения требований к памяти и статистической эффективности. Механизм разделения параметров наглядно изображен на рис. 9.5.

Для иллюстрации практического применения первых двух принципов на рис. 9.6 показано, что разреженная связность и разделение параметров кардинально улучшают эффективность линейной функции при обнаружении границ в изображении.

В случае свертки специальный вид разделения параметров наделяет слой свойством, которое называется **эквивариантностью** относительно параллельного переноса. Говорят, что функция эквивариантна, если при изменении входа выход меняется точно так же. Точнее, функция $f(x)$ эквивариантна относительно функции g,

если $f(g(x)) = g(f(x))$. В случае свертки, если g – параллельный перенос, или сдвиг входа, то функция свертки эквивариантна относительно g. Пусть, например, функция I определяет яркость в точках с целыми координатами, и пусть g – функция, отображающая одну функцию изображения в другую функцию изображения, так что $I' = g(I)$ – функция изображения, для которой $I'(x, y) = I(x - 1, y)$. Это сдвиг каждого пикселя I на одну позицию вправо. Если применить это преобразование к I, а затем выполнить свертку, то результат будет таким же, как если бы мы сначала применили свертку к I', а затем преобразование g к результату. При обработке временных рядов это означает, что свертка порождает своего рода временную шкалу, на которой показано время появления различных признаков во входных данных.

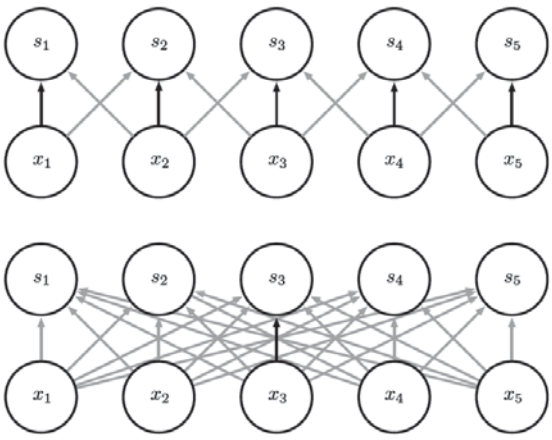

Рис. 9.5 ❖ Разделение параметров. Черными стрелками показаны связи, в которых участвует конкретный параметр в двух разных моделях. (*Вверху*) Черными стрелками показано использование центрального элемента 3-элементного ядра в сверточной модели. Благодаря разделению параметров этот параметр используется для всех элементов входа. (*Внизу*) Одиночная черная стрелка показывает использование центрального элемента матрицы весов в полносвязной модели. Здесь никакого разделения параметров нет, поэтому параметр используется только один раз

Если перенести событие на более поздний момент времени во входных данных, то на выходе появится точно такое же его представление, только позже. А при работе с изображениями свертка создает двумерную карту появления определенных признаков во входном изображении. Если переместить объект во входном изображении, то его представление на выходе переместится на такую же величину. Это бывает нужно, когда мы знаем, что некоторая функция от небольшого числа пикселей полезна при применении к нескольким участкам входа. Например, в случае обработки изображений полезно обнаруживать границы в первом слое сверточной сети. Одни и те же границы встречаются более-менее везде в изображении, поэтому имеет смысл разделять параметры по всему изображению. Но в некоторых случаях такое глобальное разделение параметров нежелательно. Например, если мы обрабатываем изображения, которые были кадрированы, так чтобы в центре оказалось лицо человека, то,

наверное, хотим выделять разные признаки в разных точках – часть сети будет обрабатывать верхнюю часть лица в поисках бровей, а другая часть – искать подбородок в нижней части лица.

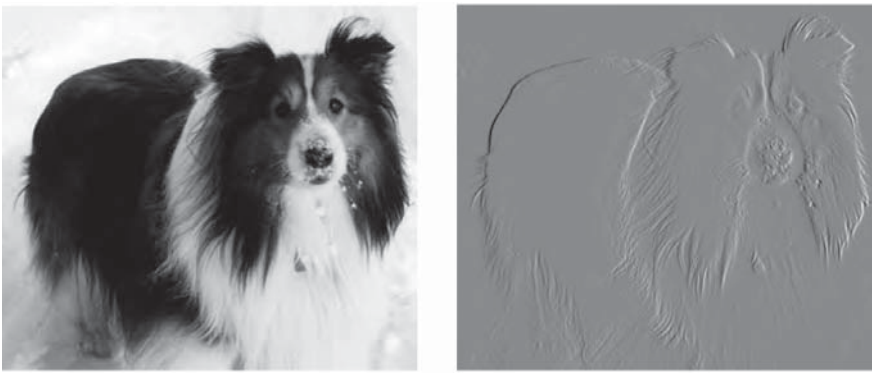

Рис. 9.6 ❖ Эффективность обнаружения границ. Правое изображение получено вычитанием из каждого пикселя исходного изображения значения пикселя слева от него. В результате мы получаем силу всех вертикальных границ во входном изображении, что бывает полезно для обнаружения объектов. Высота обоих изображений 280 пикселей. Ширина входного изображения 320 пикселей, а выходного – 319. Это преобразование можно описать как свертку с ядром, содержащим два элемента, оно требует 319 × 280 × 3 = 267 960 операций с плавающей точкой (два умножения и одно сложение на каждый выходной пиксель). Если то же самое преобразование выполнять путем перемножения матриц, то потребуется 320 × 280 × 319 × 280, т. е. больше восьми миллиардов элементов матрицы, так что с точки зрения потребления памяти свертка эффективнее такого преобразования в четыре миллиарда раз. При прямом перемножении матриц пришлось бы выполнить свыше 16 миллиардов операций с плавающей точкой, так что и с этой точки зрения свертка примерно в 60 000 раз эффективнее. Конечно, большинство элементов матрицы было бы равно нулю. Если хранить только ненулевые элементы, то в обоих случаях пришлось бы выполнить примерно одно и то же число операций с плавающей точкой. Но все равно в матрице было бы 2 × 319 × 280 = 178 640 элементов. Свертка – чрезвычайно эффективный способ описания преобразований, в которых одно и то же линейное преобразование многократно применяется к небольшим участкам изображения

Свертка не эквивариантна относительно некоторых других преобразований, например масштабирования или поворота. Для обработки таких преобразований нужны другие механизмы.

Наконец, существуют типы данных, которые нельзя обработать с помощью нейронных сетей, определяемых путем умножения на матрицу фиксированной формы. Свертка позволяет обрабатывать некоторые данные такого рода. Мы вернемся к этому вопросу в разделе 9.7.

9.3. Пулинг

Типичный слой сверточной сети состоит из трех стадий (рис. 9.7). На первой стадии слой параллельно выполняет несколько сверток и порождает множество линейных активаций. На второй стадии каждая линейная активация пропускается через нелинейную функцию активации, например функцию линейной ректификации. Эту стадию часто называют **детекторной**. На третьей стадии используется **функция пулинга** для дальнейшей модификации выхода слоя.

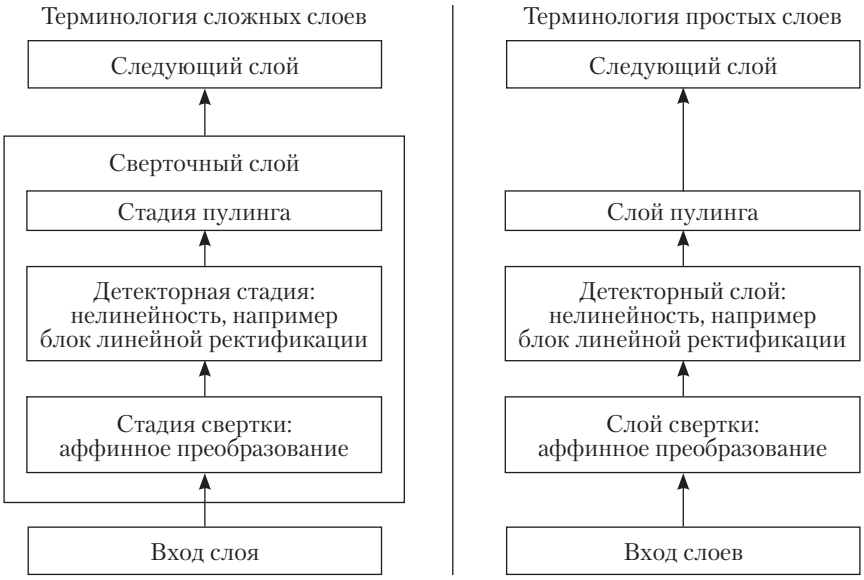

Рис. 9.7 ❖ Компоненты типичного слоя сверточной нейронной сети. Для описания таких слоев применяется двоякая терминология. (Слева) В этой терминологии сверточная сеть рассматривается как небольшой набор относительно сложных слоев, каждый из которых имеет много «стадий». При этом существует взаимно однозначное соответствие между ядерными тензорами и слоями сети. В этой книге мы в основном придерживаемся такой терминологии. (Справа) В этой терминологии сверточная сеть рассматривается как большой набор простых слоев слоев, а каждый шаг обработки считается полноправным слоем. Это означает, что не у каждого «слоя» есть параметры

Функция пулинга заменяет выход сети в некоторой точке сводной статистикой близлежащих выходов. Например, операция **max-пулинга** (Zhou and Chellappa, 1988) возвращает максимальный выход в прямоугольной окрестности. Из других употребительных функций пулинга отметим усреднение по прямоугольной окрестности, L^2-норму в прямоугольной окрестности и взвешенное среднее с весами, зависящими от расстояния до центрального пикселя.

В любом случае пулинг позволяет сделать представление приблизительно **инвариантным** относительно малых параллельных переносов входа. Инвариантность относительно параллельного переноса означает, что если сдвинуть вход на небольшую

величину, то значения большинства подвергнутых пулингу выходов не изменятся. На рис. 9.8 показан пример работы этого механизма. *Локальная инвариантность относительно параллельного переноса полезна, если нас больше интересует сам факт существования некоторого признака, а не его точное местонахождение.* Например, когда мы хотим определить, присутствует ли в изображении лицо, нам не важно положение глаз с точностью до пикселя, нужно только знать, есть ли глаз слева и глаз справа. В других ситуациях важнее сохранить местоположение признака. Например, если мы ищем угловую точку, образованную пересечением двух границ, ориентированных определенным образом, то необходимо сохранить положение границ настолько точно, чтобы можно было проверить, пересекаются ли они.

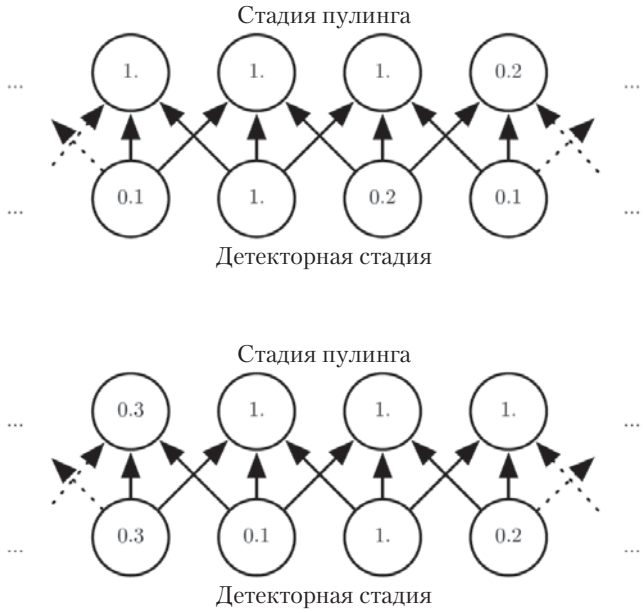

Рис. 9.8 ❖ Max-пулинг привносит инвариантность. (*Вверху*) Середина выходного слоя сверточной сети. В нижней строке показаны выходы нелинейности, а в верхней – выходы max-пулинга с шагом в один пиксель между областями пулинга, каждая из которых имеет ширину три пикселя. (*Вверху*) Та же самая сеть, сдвинутая вправо на один пиксель. В нижней строке изменились все значения, а в верхней – только половина, потому что блоки max-пулинга чувствительны лишь к максимальному значению в своей окрестности, а не точному положению этого значения

Пулинг можно рассматривать как добавление бесконечно сильного априорного предположения, что обучаемая слоем функция должна быть инвариантна к малым параллельным переносам. Если это предположение правильно, то оно может существенно улучшить статистическую эффективность сети.

Пулинг по пространственным областям порождает инвариантность к параллельным переносам, но если он производится по выходам сверток с различными параметрами, то признаки могут обучиться, к каким преобразованиям стать инвариантными (см. рис. 9.9).

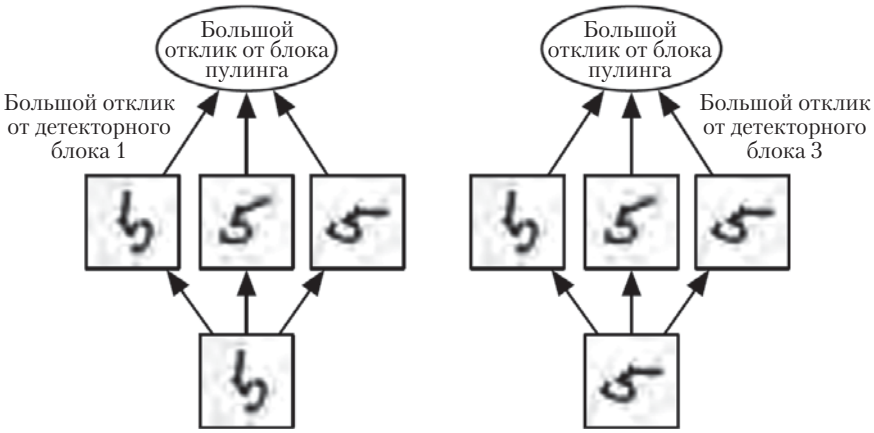

Рис. 9.9 ❖ Пример обученной инвариантности. Блок, выполняющий пулинг по нескольким признакам, обученным с разными параметрами, может обучиться инвариантности к преобразованиям входа. Здесь мы видим набор из трех обученных фильтров и блок max-пулинга, который обучился инвариантности к вращению. Все три фильтра предназначены для распознавания рукописной цифры 5. Каждый фильтр настроен на свою ориентацию пятерки. Если на входе появляется цифра 5, то соответствующий фильтр распознает ее, что даст большой отклик на детекторный блок. Тогда блок max-пулинга даст большой отклик независимо от того, какой детекторный блок был активирован. На рисунке показано, как сеть обрабатывает два разных входа, активирующих разные детекторные блоки. В обоих случаях выход блока пулинга примерно одинаков. Этот принцип используется в maxout-сетях (Goodfellow et al., 2013a) и других сверточных сетях. Max-пулинг по пространственной области обладает естественной инвариантностью к параллельным переносам; такой многоканальный подход необходим только для обучения другим преобразованиям

Поскольку пулинг агрегирует отклики по целой окрестности, количество блоков пулинга можно сделать меньшим, чем количество детекторных блоков, если агрегировать статистику по областям, отстоящим друг от друга на $k > 1$ пикселей. Пример приведен на рис. 9.10. Тем самым повышается вычислительная эффективность сети, поскольку следующему слою предстоит обработать примерно в k раз меньше входов. Если число параметров в следующем слое – функция от размера входа (например, когда следующий слой полносвязный и основан на умножении матриц), то уменьшение размера входа также может повысить статистическую эффективность и уменьшить требования к объему памяти для хранения параметров.

В большинстве задач пулинг необходим для обработки входов переменного размера. Например, если мы хотим классифицировать изображения разного размера, то код слоя классификации должен иметь фиксированный размер. Обычно это достигается за счет варьирования величины шага между областями пулинга, так чтобы слой классификации всегда получал одинаковый объем сводной статистики независимо от размера входа. Так, можно определить финальный слой пулинга в сети, так чтобы он выводил четыре сводных статистических показателя, по одному на каждый квадрант изображения, вне зависимости от размера самого изображения.

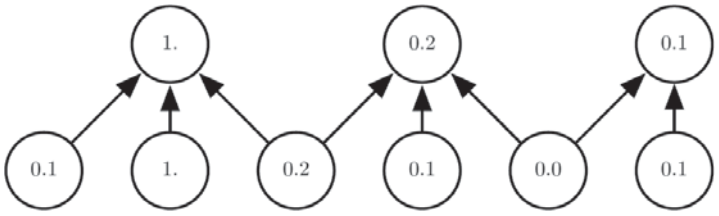

Рис. 9.10 ❖ Пулинг с понижающей передискретизацией. Здесь max-пулинг используется с пулом ширины 3 и шагом 2 между пулами. В результате размер представления уменьшается вдвое, что снижает вычислительную и статистическую нагрузки на следующий слой. Отметим, что размер самой правой области пулинга меньше остальных, но ее все равно необходимо включить, если мы не хотим игнорировать некоторые детекторные блоки

Существуют кое-какие теоретические рекомендации по выбору вида пулинга в различных ситуациях (Boureau et al., 2010). Можно также динамически агрегировать признаки, например путем выполнения алгоритма кластеризации в местах интересных признаков (Boureau et al., 2011). При таком подходе получаются различные множества областей пулинга для каждого изображения. Другой подход – обучить единую структуру пулинга и затем применять ее ко всем изображениям (Jia et al., 2012). Пулинг может внести усложнения в некоторые архитектуры нейронных сетей, где используется нисходящая информация, как, например, машины Больцмана и автокодировщики. Мы обсудим эти вопросы, когда дойдем до сетей этого типа в части III. Пулинг в сверточных машинах Больцмана представлен в разделе 20.6. Операции квазиобращения над блоками пулинга, необходимые в некоторых дифференцируемых сетях, обсуждаются в разделе 20.10.6.

Примеры полных архитектур сети для классификации с использованием свертки и пулинга показаны на рис. 9.11.

9.4. Свертка и пулинг как бесконечно сильное априорное распределение

Напомним понятие **априорного распределения вероятности**, введенное в разделе 5.2. Это распределение вероятности параметров модели, в котором закодированы наши предварительные – еще до знакомства с данными – предположения о том, какие модели считать разумными.

Априорное распределение может быть сильным или слабым в зависимости от концентрации плотности вероятности. Слабым называется априорное распределение с высокой энтропией, например нормальное распределение с большой дисперсией. При таком априорном распределении параметры могут сдвигаться в зависимости от данных более или менее свободно. У сильного априорного распределения очень низкая энтропия, как, например, у нормального распределения с малой дисперсией. Такое распределение играет более активную роль в определении конечных параметров.

В бесконечно сильном априорном распределении вероятность некоторых параметров нулевая, т. е. утверждается, что такие значения параметров запрещены вне зависимости от того, поддерживаются они данными или нет.

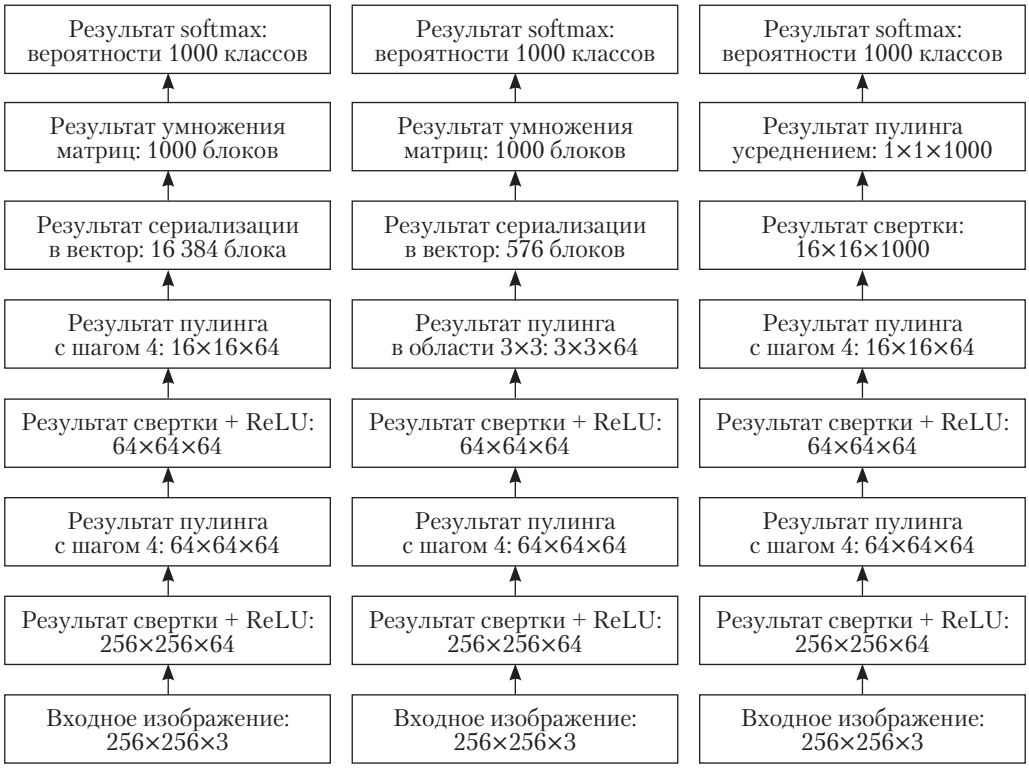

Рис. 9.11 ❖ Примеры архитектур для классификации на основе сверточных сетей. Конкретные величины шага и глубины на этом рисунке не следует рассматривать как рекомендации для практического применения; они выбраны так, чтобы рисунок поместился на страницу. Кроме того, реальные сверточные сети часто являются сильно разветвленными, а не цепными, как на этом рисунке. (*Слева*) Сверточная сеть для обработки изображений фиксированного размера. После чередования нескольких сверточных и пулинговых слоев форма тензора сверточной карты признаков меняется, чтобы выстроить все пространственные измерения в один плоский вектор. Далее следует обычный классификатор в виде сети прямого распространения, описанный в главе 6. (*В центре*) Сверточная сеть для обработки изображений переменного размера, в которой по-прежнему имеется полносвязный участок. В сети применяется операция пулинга с фиксированным числом пулов переменного размера, которая порождает вектор из 576 блоков, подаваемый на вход полносвязного участка сети. (*Справа*) Сверточная сеть, в которой вообще нет полносвязного слоя весов. Вместо этого последний сверточный слой выводит по одной карте признаков на каждый класс. Предположительно модель обучает карту тому, насколько вероятно появление каждого класса в каждой пространственной области. Единственное значение, получающееся в результате усреднения карты признаков, подается на вход softmax-классификатора в верхнем слое

Сверточную сеть можно представлять себе как полносвязную сеть с бесконечно сильным априорным распределением весов. Оно говорит, что веса некоторого скрытого слоя должны быть идентичны весам соседнего с ним слоя, но сдвинуты в простран-

стве. Априорное распределение говорит также, что веса должны быть равны 0 всюду, кроме малого рецептивного поля, состоящего из смежных блоков, поставленных в соответствие данному скрытому блоку. Короче говоря, можно считать, что свертка вводит бесконечно сильное априорное распределение вероятности параметров слоя, согласно которому обучаемая данным слоем функция допускает только локальные взаимодействия и эквивариантна относительно параллельных переносов. Аналогично пулинг вводит бесконечно сильное априорное распределение, согласно которому каждый блок должен быть инвариантен относительно малых параллельных переносов.

Разумеется, реализация сверточной сети как полносвязной сети с бесконечно сильным априорным распределением с вычислительной точки зрения была бы неимоверно расточительной. Но такая интерпретация может пролить свет на механизмы работы сверточных сетей.

Одно из ключевых открытий состоит в том, что свертка и пулинг могут стать причиной недообучения. Как любое априорное распределение, свертка и пулинг полезны, только когда предположения, выраженные этим распределением, достаточно верны. Если в задаче необходимо сохранять точную пространственную информацию, то применение пулинга по всем признакам может увеличить ошибку обучения. Некоторые архитектуры сверточных сетей (Szegedy et al., 2014a) рассчитаны на применение пулинга только к части каналов, чтобы получить как признаки с высокой степенью инвариантности, так и признаки, не подверженные недообучению в случае, когда априорное предположение об инвариантности относительно параллельных переносов неверно. Если задача подразумевает включение в состав входных данных информации из очень отдаленных областей, то априорное распределение, ассоциируемое со сверткой, может оказаться непригодным.

Еще одно важное соображение заключается в том, что при эталонном тестировании качества статистического обучения сверточные модели следует сравнивать только с другими сверточными моделями. Модели, в которых свертка не используется, смогут обучиться, даже если мы переставим все пиксели в изображении. Для многих наборов изображений существуют раздельные эталонные тесты для моделей, которые **инвариантны относительно перестановок** и должны в процессе обучения выявить концепцию топологии, и для моделей, в которые проектировщик заложил знания о пространственных связях.

9.5. Варианты базовой функции свертки

При обсуждении свертки в контексте нейронных сетей мы обычно не имеем в виду стандартную операцию дискретной свертки в том смысле, в каком она понимается в математической литературе. Практически используемые функции немного отличаются. Ниже мы детально опишем эти различия и выделим некоторые полезные свойства функций, применяемых в нейронных сетях.

Прежде всего под сверткой в нейронных сетях мы обычно понимаем операцию, состоящую из многих параллельных вычислений свертки. Связано это с тем, что свертка с одним ядром может выделить только один вид признаков, хотя и во многих местах. Обычно мы хотим, чтобы каждый слой сети выделял много разных признаков во многих местах.

Кроме того, вход обычно является не просто сеткой вещественных значений, а сеткой векторных наблюдений. Например, каждый пиксель цветного изображения состоит из

красной, зеленой и синей компонент. В многослойной сверточной сети входом второго слоя является выход первого, который обычно состоит из результатов многих разных сверток в каждой позиции. При работе с изображениями мы обычно интерпретируем вход и выход свертки как трехмерные тензоры, в которых по одному измерению находятся цветовые каналы, а по двум другим – пространственные координаты каналов. Программные реализации обычно работают в пакетном режиме, так что на самом деле используются четырехмерные тензоры, в которых четвертая ось соответствует примерам в пакете, но в нашем описании мы для простоты будем эту ось опускать.

Поскольку в сверточных сетях свертка обычно применяется к нескольким каналам, коммутативность соответствующих операций не гарантируется, даже при использовании ядра с отражением. Такие многоканальные операции коммутативны, только если число выходных каналов равно числу входных.

Предположим, что имеется четырехмерный тензор $\mathbf{K}$, элемент $K_{i,j,k,l}$ которого определяет силу связи между блоком i-го выходного канала и j-го входного канала, когда входной и выходной блоки отстоят друг от друга на k строк и l столбцов. Предположим также, что вход состоит из данных наблюдений $\mathbf{V}$, где элемент $V_{i,j,k}$ содержит значение i-го канала входного блока на пересечении j-ой строки и k-го столбца. И пусть выход $\mathbf{Z}$ имеет такой же формат, как $\mathbf{V}$. Если $\mathbf{Z}$ порожден сверткой $\mathbf{V}$ со скользящим ядром $\mathbf{K}$ без отражения $\mathbf{K}$, то

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}, \tag{9.7}$$

где суммирование производится по всем индексам l, m, n, для которых элементы тензоров под знаком суммы существуют. В линейной алгебре индексирование массива начинается с 1, что объясняет появление -1 в формуле выше. В таких языках программирования, как C и Python, индексирование начинается с 0, так что выражение даже упрощается.

Для уменьшения стоимости вычислений некоторые положения ядра иногда пропускают (ценой снижения точности выделения признаков). Мы можем рассматривать это как понижающую передискретизацию выхода полной функции свертки. Если мы хотим выбирать только каждый s-ый пиксель в каждом направлении выхода, то можем определить функцию свертки пониженного разрешения c:

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}]. \tag{9.8}$$

Будем называть s **шагом** свертки пониженного разрешения. Можно также определить разные шаги по каждому направлению движения (см. рис. 9.12).

Важное свойство любой реализации сверточной сети – возможность неявно дополнять нулями вход $\mathbf{V}$, чтобы расширить его. Без этого ширина представления уменьшается на ширину ядра без одного пикселя в каждом слое. Дополнение входа нулями позволяет управлять шириной ядра и размером выхода независимо. Не будь дополнения, мы были бы вынуждены выбирать между быстрым уменьшением пространственной протяженности сети и использованием малых ядер – то и другое значительно ограничивает выразительную мощность сети. Пример приведен на рис. 9.13.

Заслуживают упоминания три частных случая дополнения нулями. Первый – когда дополнение не используется вовсе, а ядру свертки разрешено занимать только те позиции, где ядро целиком умещается внутри изображения. В терминологии MATLAB это называется **корректной** (valid) сверткой. В таком случае каждый выходной пиксель

является функцией одного и того же числа входных, поэтому поведение выходных пикселей несколько более регулярно. Но размер выхода при этом уменьшается на каждом слое. Если ширина входного изображения равна m, а ширина ядра – k, то ширина выхода будет равна $m - k + 1$. Скорость такого сжатия может быть очень велика, если используются большие ядра. Поскольку сжатие больше 0, число сверточных слоев в сети ограничено. По мере добавления слоев пространственная протяженность сети рано или поздно снизится до 1×1, после чего новые слои нельзя считать сверточными. Другой частный случай – дополнение столькими нулями, чтобы размер выхода был равен размеру входа. В MATLAB это называется **конгруэнтной** (same) сверткой. В этом случае сеть может содержать столько сверточных слоев, сколько может поддержать оборудование, поскольку операция свертки не изменяет архитектурных возможностей следующего слоя. Однако входные пиксели в окрестности границы оказывают меньшее влияние на выходные, чем пиксели, расположенные ближе к центру. Из-за этого граничные пиксели будут недостаточно представлены в модели. Это и послужило основанием для еще одного крайнего случая, называемого в MATLAB **полной** (full) сверткой, когда добавляется столько нулей, чтобы каждый пиксель посещался k раз в каждом направлении. В результате размер выходного изображения становится равен $m + k - 1$. В таком случае выходные пиксели вблизи границы являются функциями меньшего числа входных пикселей, чем пиксели в центре. Из-за этого трудно обучить единственное ядро, которое вело бы себя хорошо во всех позициях сверточной карты признаков. Обычно оптимальная степень дополнения нулями (в терминах верности классификации на тестовом наборе) лежит между «корректной» и «конгруэнтной» сверткой.

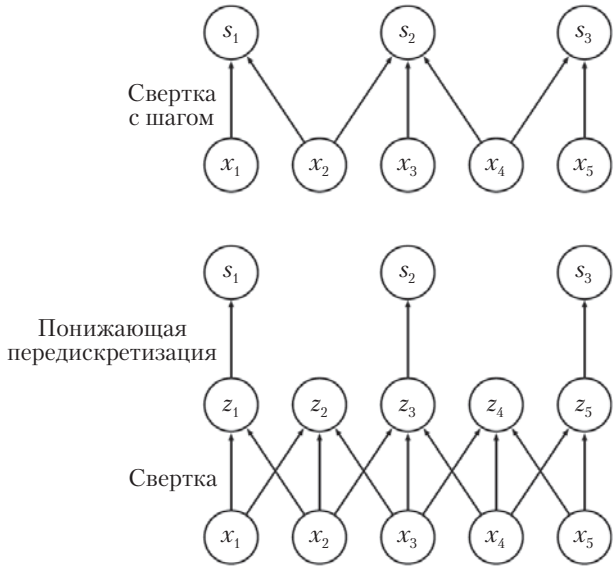

Рис. 9.12 ❖ Свертка с шагом. В этом примере шаг равен 2. (Вверху) Свертка с шагом 2, реализованная в виде одной операции. (Внизу) Свертка с шагом больше 1 пикселя математически эквивалентна свертке с шагом 1, за которой следует понижающая передискретизация. Очевидно, что такой двухэтапный подход вычислительно расточителен, поскольку многие вычисленные значения затем отбрасываются

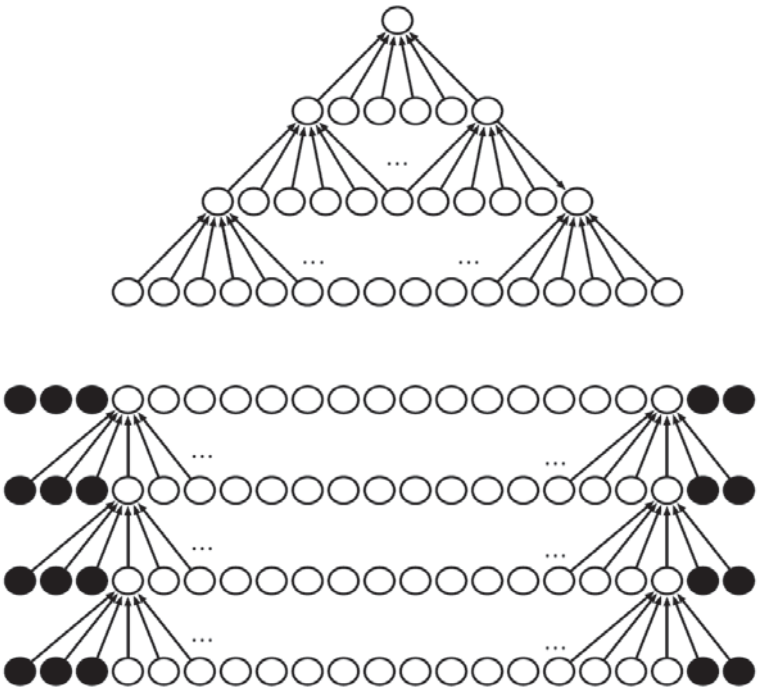

Рис. 9.13 ❖ Влияние дополнения нулями на размер сети. Рассматривается сверточная сеть с ядром ширины 6 в каждом слое. В этом примере пулинг не используется, поэтому уменьшение размера сети вызвано только самой операцией свертки. (Вверху) В этой сверточной сети нет никакого неявного дополнения нулями. Поэтому ширина представления уменьшается на пять пикселей с каждым слоем. Если ширина входа равна 16 пикселям, то в сети может быть не более трех сверточных слоев, причем по последнему переместить ядро невозможно, так что лишь два слоя можно назвать сверточными. Скорость сжатия можно уменьшить, если использовать ядра поменьше, но малые ядра обладают меньшей выразительностью, а сжатие как таковое все равно присутствует. (Снизу) Неявно добавив пять нулей в каждый слой, мы препятствуем сжатию представления. В результате глубина сверточной сети может быть любой.

Бывает, что мы хотим использовать не свертку, а локально связные слои (LeCun, 1986, 1989). В таком случае матрица смежности графа нашего МСП не изменяется, но у каждого соединения имеется собственный вес, определяемый шестимерным тензором **W**. Индексы **W** следующие: i – выходной канал, j – выходная строка, k – выходной столбец, l – входной канал, m – смещение строки внутри входа, n – смещение столбца внутри входа. Тогда линейная часть локально связного слоя описывается формулой:

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]. \tag{9.9}$$

Иногда эту операцию называют **неразделяемой сверткой** (unshared convolution), потому что она похожа на дискретную свертку с малым ядром, но без разделения параметров в разных позициях. На рис. 9.14 проведено сравнение локально связной, сверточной и полносвязной сетей.

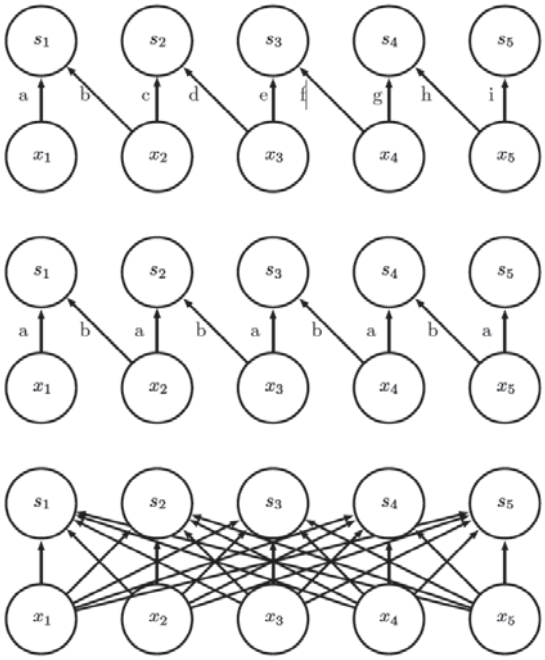

Рис. 9.14 ❖ Сравнение локальной связности, свертки и полной связности. (*Вверху*) Локально связный слой с патчем шириной два пикселя. Все ребра помечены разными буквами, т. е. с каждым ребром ассоциирован свой весовой параметр. (*В центре*) Сверточный слой с ядром шириной два пикселя. Связность этой модели точно такая же, как у локально связного слоя. Различие не в том, какие блоки взаимодействуют друг с другом, а в том, как разделяются параметры. В локальном связном слое никакого разделения параметров нет. В сверточном слое два одинаковых же веса повторно используются для всего входного слоя, это видно из повторения буквенных меток ребер. (*Внизу*) Полносвязный слой похож на локально связный в том смысле, что у каждого ребра свой собственный параметр (их слишком много, чтобы расставлять буквы на рисунке). А отличие в том, что в локальном связном слое связность ограничена

Локально связные слои полезны, если мы знаем, что каждый признак должен быть функцией небольшого участка пространства, но нет причин полагать, что один и тот же признак должен встречаться во всем пространстве. Например, если нам нужно только определить, изображено ли на картинке лицо, то требуется лишь поискать рот в нижней части изображения.

Иногда бывают полезны варианты сверточных или локально связных слоев, в которых на связность наложены дополнительные ограничения, например что каждый выходной канал i должен быть функцией лишь подмножества входных каналов l. Для этого, как правило, связывают первые m выходных каналов только с первыми n входными, следующие m выходных каналов – только со следующими n входными и т. д. (см. рис. 9.15). Поскольку моделируется взаимодействие между небольшим числом каналов, число параметров сети можно уменьшить, а значит, сократить потребление памяти, повысить статистическую эффективность и уменьшить объем вычислений,

необходимых для прямого и обратного распространений. И всех этих целей мы достигаем без уменьшения числа скрытых блоков.

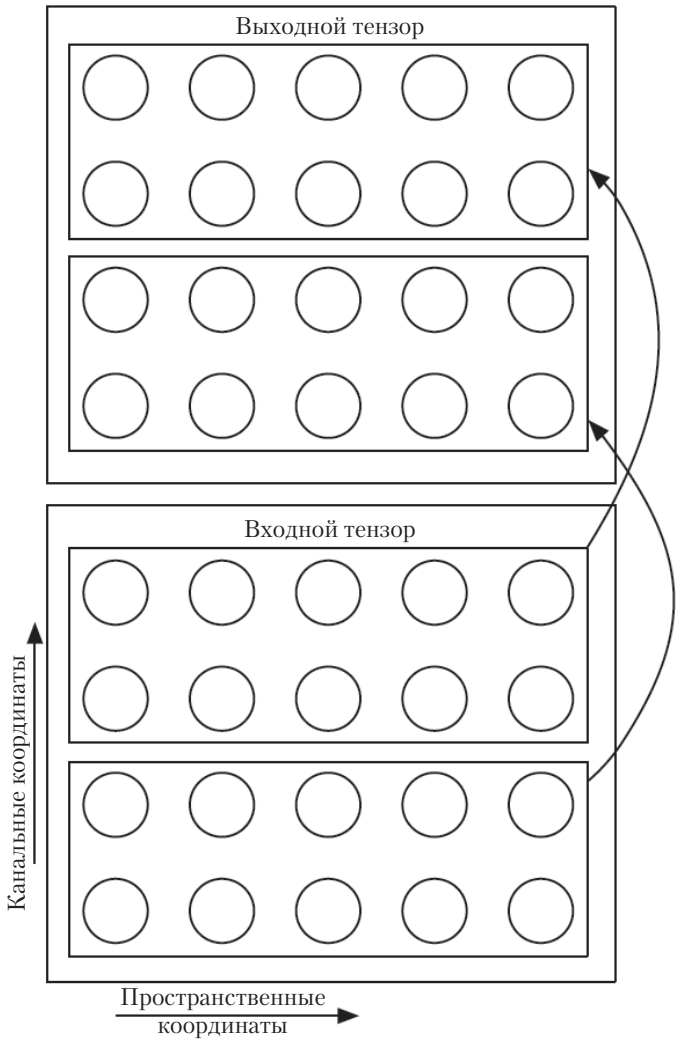

Рис. 9.15 ❖ Сверточная сеть, в которой первые два выходных канала связаны только с первыми двумя входными каналами, а следующие два выходных канала – только со следующими двумя входными

Периодическая свертка (tiled convolution) (Gregor and LeCun, 2010a; Le et al., 2010) представляет собой компромисс между сверточным и локально связным слоями. Вместо того чтобы обучать отдельный набор весов в каждой области пространства, мы обучаем набор ядер, который затем сдвигаем по пространству как единое целое. Это означает, что в соседних областях фильтры будут разные, как в локально связном слое, но требования к объему памяти для хранения параметров возрастают лишь пропорционально размеру набора ядер, а не как при хранении всей выходной

карты признаков. На рис. 9.16 проведено сравнение локально связных слоев, периодической и стандартной сверток.

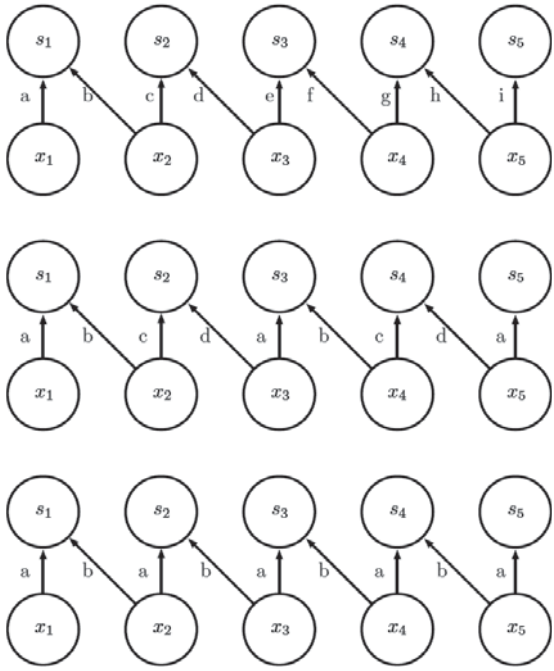

Рис. 9.16 ❖ Сравнение локально связных слоев, периодической свертки и стандартной свертки. Во всех трех случаях при использовании ядра одного размера набор связей между блоками один и тот же. На этом рисунке предполагается, что ширина ядра составляет два пикселя. Различие между методами – в разделении параметров. (*Вверху*) В локально связном слое параметры не разделяются вовсе. Все связи помечены разными буквами, т. е. у каждой связи свой вес. (*В центре*) В случае периодической свертки имеется набор из t разных ядер. В данном случае $t = 2$. Ребра первого ядра помечены буквами «a» и «b», а ребра второго – буквами «c» и «d». При смещении в выходном слое на один пиксель вправо мы переходим к использованию другого ядра. Это означает, что, как и в локально связном слое, у соседних выходных блоков параметры разные. Но, в отличие от локально связного слоя, после перебора всех t имеющихся ядер мы снова возвращаемся к первому. Два выходных блока, разделенных числом шагов, кратным t, разделяют общие параметры. (*Внизу*) Традиционная свертка эквивалентна периодической с $t = 1$. Существует всего одно ядро, которое применяется во всех точках. На рисунке это следует из того, что все ребра помечены буквами «a» и «b»

Чтобы определить периодическую свертку алгебраически, рассмотрим шестимерный тензор **K**, два измерения которого соответствуют различным позициям в выходной карте. Вместо отдельного индекса для каждой позиции выходной карты индексы будут циклически пробегать множество t различных положений группы ядер в каждом направлении. Если t равно ширине выхода, то мы получаем локально связный слой.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j\%t+1,k\%t+1}, \tag{9.10}$$

где знак процента обозначает операцию деления по модулю: $t\%t = 0$, $(t+1)\%t = 1$ и т. д. Эта формула легко обобщается на различные периоды по каждому направлению.

Локально связные слои и слои с периодической сверткой интересно взаимодействуют с max-пулингом: детекторные блоки таких слоев управляются разными фильтрами. Если эти фильтры обучены обнаруживать различные преобразованные варианты одних и тех же базовых признаков, то блоки max-пулинга оказываются инвариантны относительно обученных преобразований (см. рис. 9.9). В сверточные слои изначально «зашита» инвариантность относительно параллельного переноса.

Для реализации сверточной сети обычно необходимы и другие операции, помимо свертки. Для обучения нужно уметь вычислять градиент относительно ядра, если известен градиент относительно выходов. В некоторых простых случаях эту операцию можно выполнить с помощью свертки, но большинство интересных случаев, в т. ч. свертка с шагом 1, не обладает этим свойством.

Напомним, что свертка – это линейная операция и потому может быть описана как умножение на матрицу (если предварительно вытянуть входной тензор в плоский вектор). Соответствующая матрица является функцией от ядра свертки. Эта матрица разреженная, и каждый элемент ядра копируется в несколько элементов матрицы. Такое представление поможет нам вывести некоторые дополнительные операции, необходимые для реализации сверточной сети.

Одна из таких операций – умножение на матрицу, транспонированную к матрице, определяемой сверткой. Эта операция нужна для обратного распространения ошибки через сверточный слой, а стало быть, необходима для обучения сверточных сетей, содержащих более одного скрытого слоя. Она же встречается, когда требуется реконструировать видимые блоки по скрытым (Simard et al., 1992). Реконструкция видимых блоков часто используется в моделях, описанных в третьей части книги: автокодировщики, ограниченные машины Больцмана и разреженное кодирование. Транспонирование свертки необходимо для построения сверточных вариантов таких моделей. Как и операцию градиента относительно ядра, эту операцию над градиентом входа иногда удается выполнить с помощью свертки, но в общем случае требуется реализовать третью операцию. Следует соблюдать осторожность при координировании операции транспонирования с прямым распространением. Размер выхода, возвращаемого операцией транспонирования, зависит от стратегии дополнения нулями и шага операции прямого распространения, а также от размера выходной карты прямого распространения. В некоторых случаях разные размеры входа прямого распространения могут давать одинаковый размер выхода, поэтому операции транспонирования нужно явно указать размер первоначального входа.

Этих трех операций – свертка, обратное распространение от выхода к весам и обратное распространение от выхода к входам – достаточно для вычисления всех градиентов, необходимых для обучения сверточной сети прямого распространения любой глубины, а также для обучения сверточной сети с функциями реконструкции, основанными на транспонировании свертки. Полный вывод уравнений в общем многомерном случае с несколькими примерами см. в работе Goodfellow (2010). Чтобы вы могли составить представление о том, как эти уравнения работают, рассмотрим двумерный случай с одним примером.

Допустим, мы хотим обучить сверточную сеть, в которой имеется свертка с шагом s группы ядер $\mathbf{K}$, применяемых к многоканальному изображению $\mathbf{V}$, опреде-

ленная функцией $c(\mathbf{K}, \mathbf{V}, s)$, как в формуле (9.8). Предположим, что требуется минимизировать некую функцию потерь $J(\mathbf{V}, \mathbf{K})$. На этапе прямого распространения нам нужно будет использовать саму функцию c, чтобы вывести $\mathbf{Z}$, который затем распространяется по остальной части сети и применяется для вычисления функции стоимости J. На этапе обратного распространения мы получим тензор $\mathbf{G}$ – такой, что $G_{i,j,k} = (\partial / \partial Z_{i,j,k}) J(\mathbf{V}, \mathbf{K})$.

Для обучения сети нам потребуется вычислить производные по весам в ядре. Для этого можно воспользоваться функцией

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k,(n-1)\times s+l}. \tag{9.11}$$

Если это не нижний слой сети, то потребуется также вычислить градиент по $\mathbf{V}$ для обратного распространения ошибки. Для этого воспользуемся функцией

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) \tag{9.12}$$

$$= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)\times s+p=k}} \sum_{q} K_{q,i,m,p} G_{q,l,n}. \tag{9.13}$$

Автокодировщики, описанные в главе 14, – это сети прямого распространения, обученные копировать вход в выход. Простой пример дает метод главных компонент PCA, который копирует свой вход $\boldsymbol{x}$ в приближенную реконструкцию $\boldsymbol{r}$, применяя функцию $\boldsymbol{W}^\top \boldsymbol{W}\boldsymbol{x}$. В более общих автокодировщиках тоже часто используется умножение на транспонированную матрицу весов, как в PCA. Чтобы превратить такие модели в сверточные, мы можем воспользоваться функцией h, которая транспонирует операцию свертки. Пусть имеются скрытые блоки $\mathbf{H}$ в том же формате, что и $\mathbf{Z}$; определим преобразование реконструкции в виде

$$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s). \tag{9.14}$$

Для обучения автокодировщика получим градиент относительно $\mathbf{R}$ в виде тензора $\mathbf{E}$. Для обучения декодера нужно получить градиент относительно $\mathbf{K}$. Он имеет вид $g(\mathbf{H}, \mathbf{E}, s)$. Для обучения кодера нужен градиент относительно $\mathbf{H}$. Он имеет вид $c(\mathbf{K}, \mathbf{E}, s)$. Можно также продифференцировать g, используя c и h, но эти операции не нужны для алгоритма обратного распространения в стандартных архитектурах сети.

В общем случае для преобразования входов в выходы в сверточном слое используются не только линейные операции. К каждому выходу еще прибавляется смещение, и лишь потом применяется нелинейность. Тогда возникает вопрос, как разделить параметры между смещениями. Для локально связных слоев естественно приписать каждому блоку свое смещение, а для периодической свертки – разделять смещения с таким же периодом, как и ядра. В сверточных слоях, как правило, используют одно смещение на выходной канал и разделяют его между всеми позициями в пределах каждой выходной карты. Но если размер входа известен и фиксирован, то можно также обучить отдельные смещения в каждой позиции выходной карты. Это несколько снижает статистическую эффективность модели, но позволяет ей корректировать различия между статистиками изображения в разных позициях. Например, если применяется неявное дополнение нулями, то детекторные блоки на границе изображения получают меньше информации от входов и могут нуждаться в больших смещениях.

9.6. Структурированный выход

Сверточные сети можно использовать для вывода многомерного структурированного объекта, а не просто для предсказания метки класса в случае классификации или вещественного значения в случае регрессии. Обычно таким объектом является тензор, порождаемый стандартным сверточным слоем. Например, модель может порождать тензор **S**, в котором $S_{i,j,k}$ – вероятность того, что входной пиксель в позиции (j, k) принадлежит классу i. Это позволяет модели пометить каждый пиксель изображения и нарисовать точные маски, выделяющие контуры отдельных объектов.

Часто возникает проблема из-за того, что выходная плоскость меньше входной, как показано на рис. 9.13. В архитектурах, которые обычно применяются для классификации одного объекта в изображении, уменьшение числа пространственных измерений сети связано прежде всего с использованием слоев пулинга с большим шагом. Чтобы вывести карту примерно такого же размера, как вход, можно вообще отказаться от пулинга (Jain et al., 2007). Другая стратегия – порождать сетку меток более низкого разрешения (Pinheiro and Collobert, 2014, 2015). Наконец, в принципе, можно использовать оператор пулинга с единичным шагом.

Одна из стратегий попиксельной пометки изображений – взять какую-нибудь начальную гипотезу о метках, а затем уточнять ее, используя взаимодействия между соседними пикселями. Многократное повторение шага уточнения соответствует использованию одинаковых сверток на каждом этапе с разделением весов между последними слоями глубокой сети (Jain et al., 2007). В результате последовательность вычислений, выполняемых соседними сверточными слоями с весами, разделяемыми между слоями, становится специальным видом рекуррентной сети (Pinheiro and Collobert, 2014, 2015). На рис. 9.17 показана архитектура такой рекуррентной сверточной сети.

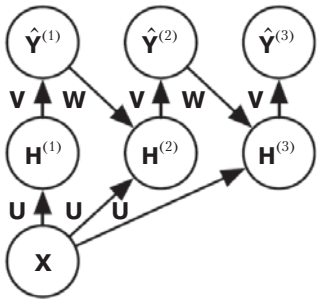

Рис. 9.17 ❖ Пример рекуррентной сверточной сети для пометки пикселей. Входом является тензор изображения **X**, оси которого соответствуют строкам, столбцам и каналам (красный, зеленый, синий) изображения. Цель – построить тензор меток $\hat{Y}$, содержащий распределение вероятности меток каждого пикселя. Оси этого тензора соответствуют строкам, столбцам и классам. Вместо того чтобы выводить $\hat{Y}$ за один присест, рекуррентная сеть итеративно уточняет оценку $\hat{Y}$, используя предыдущую оценку как исходные данные для вычисления новой. Для каждого приближения к оценке используются одни и те же параметры, количество уточнений может быть любым. На каждом шаге используется тензор сверточных ядер **U** для вычисления скрытого представления входного изображения. Ядерный тензор **V** используется для порождения оценки меток при заданных скрытых значениях. На всех шагах, кроме первого, ядра **W** сворачиваются с $\hat{Y}$ для вычисления входа скрытого слоя. На первом шаге этот член заменяется нулем. Поскольку параметры на всех шагах одинаковы, это пример рекуррентной сети в смысле главы 10

Получив предсказания для всех пикселей, мы можем применить различные методы для последующей обработки предсказания, чтобы сегментировать изображение на отдельные участки (Briggman et al., 2009; Turaga et al., 2010; Farabet et al., 2013). Общая идея в том, что большие группы смежных пикселей предположительно ассоциированы с одной и той же меткой. С помощью графических моделей можно описать вероятностные связи между соседними пикселями. Или обучить сверточную сеть максимизировать аппроксимацию целевой функции графической модели (Ning et al., 2005; Thompson et al., 2014).

9.7. Типы данных

Данные, используемые в сверточной сети, обычно состоят из нескольких каналов, каждый из которых содержит наблюдение какой-то величины в определенной точке пространства или в определенный момент времени. Примеры типов данных разной размерности с разным числом каналов приведены в табл. 9.1.

Таблица 9.1. Примеры форматов данных в сверточных сетях

	Одноканальные	Многоканальные
1D	Аудиосигналы: ось свертки соответствует времени. Мы дискретизируем время и измеряем амплитуду сигнала один раз в каждом временном интервале	Данные анимации «скелета»: анимации трехмерного отрисовываемого компьютером персонажа генерируются путем изменения положения «скелета» со временем. В каждый момент положение скелета описывается углами сочленения костей в каждом суставе. Каждый канал данных, подаваемых на вход сверточной сети, представляет угол относительно одной оси одного сустава
2D	Аудиоданные, предварительно обработанные с помощью преобразования Фурье. Мы можем преобразовать аудиосигнал в двумерный тензор, строки которого соответствуют частотам, а столбцы – моментам времени. Применение свертки по времени делает модель эквивариантной относительно временных сдвигов. Применение свертки по оси частот делает модель эквивариантной относительно частоты, т. е. одна и та же мелодия в разных октавах порождает на входе сети одно и то же представление, но с разной высотой	Данные цветного изображения: один канал содержит красные пиксели, другой – зеленые, третий – синие. Ядро свертки сдвигается по двум осям изображения, обеспечивая эквивариантность относительно параллельного переноса в обоих направлениях
3D	Объемные данные: типичным источником таких данных являются технологии медицинской интроскопии, например компьютерной томографии	Данные цветного видео: одна ось соответствует времени, другая – высоте кадра, третья – ширине кадра

Пример применения сверточной сети к обработке видео см. в работе Chen et al. (2010).

До сих пор мы рассматривали только случай, когда все примеры в обучающем и тестовом наборах данных имели одинаковые пространственные размеры. Но сверточные сети способны также обрабатывать входные данные с разной пространственной протяженностью. Такие данные вообще невозможно представить с помощью традиционных нейронных сетей, основанных на умножении матриц. И это убедительная причина использовать сверточные сети даже тогда, когда вычислительная стоимость и переобучение не составляют проблемы.

Рассмотрим, к примеру, собрание изображений, в котором у всех изображений разная ширина и высота. Не понятно, как можно смоделировать такие входные данные с помощью матрицы весов фиксированного размера. А со сверткой никаких проблем не возникает; ядро просто применяется разное число раз в зависимости от размера входа, и выход операции свертки масштабируется соответственно. Свертку можно рассматривать как умножение на матрицу; одно и то же ядро индуцирует дважды блочно-циркулянтные матрицы разного размера, зависящего от размера входа. Иногда не только вход, но и выход сети может иметь разный размер, например если мы хотим сопоставить метку класса каждому входному пикселю. В этом случае ничего специально проектировать не нужно. А бывает и так, что сеть должна порождать выход фиксированного размера, например если мы хотим сопоставить одну метку класса всему изображению. Тогда следует предпринять дополнительные действия, например вставить слой пулинга, в котором области агрегирования масштабируются пропорционально размеру входа, чтобы количество выходов пулинга было фиксированным. Примеры такой стратегии показаны на рис. 9.11.

Отметим, что использование свертки для обработки входных данных переменного размера имеет смысл, только если переменность размера вызвана тем, что вход содержит разный объем данных наблюдений одной и той же сущности: аудиозаписи разной продолжительности, космические снимки разной ширины и т. д. Свертка не имеет смысла, если размер входных данных изменяется, потому что включены разнородные наблюдения. Например, если обрабатываются заявления о поступлении в колледж и признаками являются оценки в аттестате и результаты экзаменов, но не все поступающие сдают экзамены, то бессмысленно сворачивать одни и те же веса с признаками, соответствующими оценкам и результатам экзаменов.

9.8. Эффективные алгоритмы свертки

В современных приложениях сверточных сетей часто участвуют сети, содержащие более миллиона блоков. Для работы с ними необходимы эффективные реализации, задействующие средства распараллеливания вычислений (см. раздел 12.1). Но во многих случаях работу можно ускорить, выбрав подходящий алгоритм свертки.

Свертка эквивалентна переводу входа и ядра в частотную область с помощью преобразования Фурье, поточечному перемножению двух сигналов и возврату во временную область с помощью обратного преобразования Фурье. При определенном размере задачи это может оказаться быстрее наивной реализации дискретной свертки в лоб.

Если d-мерное ядро можно представить в виде внешнего произведения d векторов, по одному на каждое измерение, то ядро называется **сепарабельным**. Для сепарабельных ядер наивная свертка неэффективна. Она эквивалентна композиции d одномерных сверток с каждым из этих векторов. Это гораздо быстрее вычисления d-мерной свертки с их внешним произведением. Кроме того, для представления ядра в виде векторов требуется меньше параметров. Если ядро состоит из w элементов в каждом направлении, то для наивной многомерной свертки потребуется время $O(w^d)$ и столько же места в памяти для хранения параметров, тогда как для сепарабельной свертки нужно время и память порядка $O(w \times d)$. Разумеется, не всякую свертку можно представить подобным образом.

Поиск более быстрых способов вычислить свертку точно или приближенно, не принося в жертву верности модели, – область активных исследований. Даже мето-

ды, улучшающие эффективность одного лишь прямого распространения, уже полезны, потому что в коммерческих системах обычно больше ресурсов расходуется не на обучение, а на развертывание сети.

9.9. Случайные признаки и признаки, обученные без учителя

Обычно самой дорогостоящей частью обучения сверточной сети является обучение признаков. Выходной слой, как правило, обходится относительно недорого, потому что ему на вход подается небольшое число признаков, прошедших несколько слоев пулинга. Если производится обучение с учителем методом градиентного спуска, то на каждом шаге необходимо выполнить полный цикл прямого и обратного распространений по всей сети. Один из способов уменьшить стоимость обучения сверточной сети – использовать признаки, для которых не применялось обучение с учителем.

Есть три основные стратегии получения сверточных ядер без обучения с учителем. Первый – просто инициализировать случайным образом. Второй – спроектировать вручную, настроив каждое ядро на обнаружение границ определенной ориентации или в определенном масштабе. Наконец, можно обучать ядра без учителя. Например, в работе Coates et al. (2011) кластеризация методом k средних применена к малым патчам изображения, а затем каждый обученный центроид использовался в качестве ядра свертки. В части III мы опишем много других подходов к обучению без учителя. Обучение признаков без учителя позволяет определять их отдельно от слоя классификации, занимающего верхнее место в архитектуре. Следовательно, можно выделить признаки для всего обучающего набора только один раз, по существу построив новый обучающий набор для последнего слоя. Тогда обучение последнего слоя часто оказывается задачей выпуклой оптимизации в предположении, что этот слой реализует логистическую регрессию, метод опорных векторов или что-то подобное.

Случайные фильтры нередко работают на удивление хорошо в сверточных сетях (Jarrett et al., 2009; Saxe et al., 2011; Pinto et al., 2011; Cox and Pinto, 2011). В работе Saxe et al. (2011) показано, что слои, состоящие из свертки, за которой следует пулинг, естественно становятся частотно-избирательными и инвариантными относительно параллельного переноса, если им приписать случайные веса. Авторы предлагают следующий дешевый способ выбора архитектуры сверточной сети: сначала оценить качество нескольких архитектур, обучив только последний слой, а затем взять лучшую архитектуру и обучить ее целиком, применив более дорогой способ.

Промежуточное решение – обучить признаки, но не использовать методов, требующих полного прямого и обратного распространений на каждом шаге вычисления градиента. Как и в многослойных перцептронах, мы применяем жадное послойное предобучение, чтобы обучить первый слой отдельно, затем выделяем все признаки только из первого слоя, потом обучаем отдельно второй слой, зная эти признаки, и т. д. В главе 8 мы описали, как выполняется жадное послойное предобучение с учителем, а в части III обобщим эту идею на жадное послойное предобучение, применяя в каждом слое критерий, не требующий учителя. Канонический пример жадного послойного предобучения сверточной модели дает сверточная глубокая сеть доверия (Lee et al., 2009). Сверточные сети дают возможность продвинуть стратегию предобучения на шаг дальше, чем было возможно в многослойных перцептронах. Вместо обучения

всего сверточного слоя за раз мы можем обучить модель небольшого патча, как сделано в работе Coates et al. (2011) с применением метода k средних. Затем параметры этой обученной на патче модели можно использовать для определения ядер сверточного слоя. Это означает, что для обучения сверточной сети можно применить обучение без учителя, *даже не используя в процессе обучения свертку*. При таком подходе можно обучать очень большие модели, а стоимость вычисления будет высока только на этапе вывода (Ranzato et al., 2007b; Jarrett et al., 2009; Kavukcuoglu et al., 2010; Coates et al., 2013). Эта идея была очень популярна с 2007 по 2013 год, когда размеченные наборы данных были невелики, а вычислительные ресурсы ограничены. Сегодня большинство сверточных сетей обучают в режиме чистого обучения с учителем, производя на каждой итерации полное прямое и обратное распространения по всей сети.

Как и в других подходах к предобучению без учителя, трудно разделить причины некоторых достоинств этой методики. Предобучение без учителя может как обеспечить частичную регуляризацию, по сравнению с обучением с учителем, так и просто дать возможность обучать гораздо более масштабные архитектуры вследствие снижения вычислительной стоимости правила обучения.

9.10. Нейробиологические основания сверточных сетей

Сверточные сети – пожалуй, самый яркий пример успешного применения биотехнологического искусственного интеллекта. Хотя на них оказали влияние и многие другие научные дисциплины, некоторые ключевые принципы были почерпнуты из нейробиологии.

История сверточных сетей начинается с нейробиологических экспериментов, поставленных задолго до создания соответствующих компьютерных моделей. Нейрофизиологи Давид Хубель и Торстен Визель в течение нескольких лет совместно установили большинство основных фактов, касающихся работы зрительной системы млекопитающих (Hubel and Wiesel, 1959, 1962, 1968). За свои достижения они были отмечены Нобелевской премией. Их открытия, оказавшие огромное влияние на современные модели глубокого обучения, были основаны на регистрации активности отдельных нейронов в мозге кошек. Они наблюдали, как нейроны реагируют на изображения, проецируемые точно на определенные участки экрана, расположенного перед кошкой. Важнейшее открытие состояло в том, что нейроны первичных зрительных центров сильнее реагируют на очень специфические зрительные паттерны, например точно ориентированные полоски, и гораздо слабее – на все остальные паттерны.

Их работа помогла охарактеризовать многие аспекты функционирования мозга, выходящие за рамки этой книги. С точки зрения глубокого обучения, нас интересует в основном упрощенная, схематическая картина.

И в этой упрощенной картине мы сосредоточимся на области мозга, которая называется зоной V1, или **первичной зрительной корой**. Это первая область мозга, которая начинает значимую обработку зрительной информации. Не вдаваясь в детали, скажем, что изображение формируется благодаря попаданию в глаз света, который стимулирует сетчатку, светочувствительный орган, составляющий внутреннюю оболочку глаза. Нейроны сетчатки выполняют простую предобработку изображения,

но не слишком сильно изменяют способ его представления. Затем изображение по зрительному нерву поступает в область мозга, которая называется *латеральным коленчатым телом*. Главная интересующая нас задача обоих этих анатомических образований – передать сигнал из глаза в зону V1, расположенную на затылке.

Слой сверточной сети улавливает три свойства зоны V1:

1) зона V1 организована в виде пространственной карты. Она имеет двумерную структуру, повторяющую структуру изображения на сетчатке. Так, свет, падающий на верхнюю половину сетчатки, воздействует только на соответствующую половину зоны V1. Сверточная сеть улавливает это свойство, поскольку ее признаки определены в терминах двумерных карт;
2) зона V1 состоит из большого числа **простых клеток**. Активность клетки можно до некоторой степени охарактеризовать линейной функцией изображения в малом пространственно локализованном рецептивном поле. Детекторные блоки сверточной сети призваны имитировать именно эти свойства простых клеток;
3) в зоне V1 имеется также много **сложных клеток**. Они реагируют на признаки, похожие на детектируемые простыми клетками, но инвариантны относительно небольших изменений в положении признаков. Отсюда берут начало пулинговые блоки сверточных сетей. Сложные клетки инвариантны также относительно некоторых изменений освещения, которые невозможно уловить с помощью простого агрегирования по пространственным областям. Эти виды инвариантности стали причиной некоторых стратегий межканального пулинга в сверточных сетях, например maxout-блоков (Goodfellow et al., 2013a).

Хотя мы знаем в основном о зоне V1, общее мнение склоняется к тому, что те же базовые принципы применимы и к другим частям зрительной системы. В нашем упрощенном представлении базовая стратегия детектирования, сопровождаемая пулингом, снова и снова применяется по мере продвижения вглубь мозга. Пройдя через многие анатомические уровни мозга, мы наконец обнаруживаем клетки, которые реагируют на специфические концепции и инвариантны относительно многих преобразований входной информации. Эти клетки получили название «бабушкиных клеток»[1], напоминающее о том, что может существовать нейрон, который активируется, когда человек видит изображение своей бабушки, вне зависимости от того, расположено оно справа или слева в поле зрения, содержит только увеличенное лицо или всю фигуру, ярко освещено или находится в тени и т. д.

Доказано, что такие бабушкины клетки действительно существуют в мозге человека, в области, которая называется *медиальной височной долей* (Quiroga et al., 2005). Ученые проверяли, какие нейроны реагируют на фотографии известных личностей. Обнаружился так называемый «нейрон Холли Берри», который активируется концепцией этой актрисы. Этот нейрон возбуждается, когда человек видит фотографию Холли Берри, рисунок Холли Берри или даже текст со словами «Холли Берри». Разумеется, в самой Холли Берри нет ничего особенного; другие нейроны реагируют на присутствие Билла Клинтона, Дженнифер Энистон и т. д.

Нейроны медиальной височной доли несколько более общие, чем современные сверточные сети, которые не могут автоматически обобщаться для идентификации

---

[1] Официальное русскоязычное название — «нейроны графических объектов», или «афферентно-инвариантные нейроны объектов». – *Прим. перев.*

человека или объекта, увидев его имя или название. Ближайшим аналогом последнего слоя признаков сверточной сети является область мозга, называемая *инферотемпоральной корой* (IT). При рассматривании объекта информация попадает сначала на сетчатку, проходит через латеральное коленчатое тело в зону V1, затем в V2, потом в V4 и, наконец, в IT. Это происходит в течение первых 100 мс после попадания объекта в поле зрения. Если человек получает возможность смотреть на объект дольше, то информация начинает течь в обратном направлении, т. к. мозг использует нисходящую обратную связь, чтобы обновить активации в областях нижних уровней. Но если прервать взгляд и понаблюдать только за частотой пульсации в результате первых 100 мс, отведенных в основном на активацию прямой связи, то зона IT оказывается похожей на сверточную сеть. Сверточные сети могут предсказать частоту пульсации IT и работают аналогично человеку (с его временными ограничениями) при решении задач распознавания объектов (DiCarlo, 2013).

Вместе с тем существует много различий между сверточными сетями и зрительной системой млекопитающих. Некоторые из них хорошо известны компьютерным нейробиологам, но выходят за рамки книги. Другие пока неизвестны, потому что на многие вопросы о зрительной системе млекопитающих еще нет ответов. Приведем лишь краткий перечень.

- У человеческого глаза очень низкое разрешение всюду, кроме небольшого пятна, называемого **центральной ямкой**, которая видит область размером примерно с ноготь большого пальца на расстоянии вытянутой руки. Нам кажется, что мы видим всю сцену с высокой разрешающей способностью, но на самом деле это иллюзия, создаваемая подсознательной частью мозга, которая сшивает много фрагментов малых областей. Большинство сверточных сетей получает на входе большие фотографии высокого разрешения. Человеческий мозг вынуждает глаза совершить несколько быстрых скачкообразных движений **(саккад)** для рассматривания самых выделяющихся или относящихся к задаче частей сцены. Включение аналогичных механизмов внимания в модели глубокого обучения – направление активных исследований. В контексте глубокого обучения механизмы внимания добились наибольшего успеха при обработке естественных языков (см. раздел 12.4.5.1). Было разработано несколько моделей зрения с механизмами, аналогичными центральной ямке, но пока они не могут претендовать на роль доминирующего подхода (Larochelle and Hinton, 2010; Denil et al., 2012).
- Зрительная система человека интегрирована с другими органами чувств, например слухом, а также такими факторами, как наше настроение и мысли. Сверточные сети пока относятся только к зрению.
- Зрительная система человека отвечает далеко не только за распознавание объектов. Она способна понимать целые сцены, включающие много объектов и связей между ними, и умеет обрабатывать сложную трехмерную геометрическую информацию, без чего наше тело не могло бы взаимодействовать с окружающим миром. Были попытки применить сверточные сети к решению таких задач, но пока эти исследования пребывают в зачаточном состоянии.
- Даже такие простые области мозга, как зона V1, сильно зависят от обратной связи с более высокими уровнями. Применению обратной связи в нейронных сетях посвящено много работ, но пока с ее помощью не удалось достичь существенного улучшения.

- Хотя частота пульсации в зоне IT отражает примерно ту же информацию, что признаки сверточной сети, не ясно, насколько похожи промежуточные вычисления. Вероятно, в мозгу используются совершенно другие функции активации и пулинга. Активацию отдельного нейрона вряд ли можно хорошо охарактеризовать откликом одного линейного фильтра. Недавняя модель зоны V1 включает четыре квадратичных фильтра для каждого нейрона (Rust et al., 2005). Вообще, наше схематичное разделение на «простые» и «сложные» клетки может отражать несуществующее различие; возможно, что простые и сложные клетки – это один и тот же вид клеток, но их «параметры» допускают бесконечную градацию поведения – от «простого» до «сложного».

Отметим также, что нейробиология мало что говорит о том, как *обучать* сверточные сети. Структуры моделей с разделением параметров между несколькими пространственными областями восходят еще к ранним коннекционистским моделям зрения (Marr and Poggio, 1976), но в этих моделях не использовались современные алгоритмы обратного распространения и градиентного спуска. Так, неокогнитрон (Fukushima, 1980) включал большинство архитектурных элементов современной сверточной сети, но опирался на алгоритм послойной кластеризации без учителя.

В работе Lang and Hinton (1988) впервые использовалось обратное распространение для обучения **нейронных сетей с временной задержкой** (time-delay neural network – TDNN). В современной терминологии TDNN – это одномерная сверточная сеть в применении к временным рядам. Применение обратного распространения в таких моделях не основано ни на каких нейробиологических наблюдениях, и некоторые ученые считают его биологически неправдоподобным. На волне успеха обучения TDNN на основе обратного распространения в работе LeCun et al. (1989) была разработана современная сверточная сеть путем применения того же алгоритма обучения к двумерным сверткам изображений.

До сих пор мы говорили, что простые клетки приблизительно линейны и избирательны к некоторым признакам, что сложные клетки в большей степени нелинейны и приобретают инвариантность относительно некоторых преобразований признаков, найденных простыми клетками, и что наличие нескольких уровней, чередующих избирательность и инвариантность, возможно, приводит к бабушкиным клеткам, распознающим конкретные явления. Мы еще точно не описали, что могут обнаруживать эти индивидуальные клетки. В глубокой нелинейной сети трудно понять функции отдельных клеток. Простые клетки первого уровня проанализировать легче, потому что их отклик описывается линейной функцией. В искусственной нейронной сети мы можем просто показать изображение ядра свертки и увидеть, на что реагирует соответствующий канал сверточного слоя. В биологической нейронной сети у нас нет доступа к самим весам. Вместо этого мы подводим электрод к нейрону, помещаем несколько примеров белого шума перед сетчаткой животного и регистрируем, какие примеры приводят к возбуждению нейрона. Затем мы подгоняем к этим откликам линейную модель, чтобы получить аппроксимацию весов нейронов.

Этот подход, получивший название **обратной корреляции** (Ringach and Shapley, 2004), показывает, что веса большинства клеток зоны V1 описываются **функциями Габора**. Функция Габора описывает вес в точке двумерного изображения. Можно считать изображение функцией координат на двумерной плоскости, $I(x, y)$. Аналогично простую клетку можно рассматривать как выборку из изображения во множе-

стве позиций, определяемую множествами абсцисс $\mathbb{X}$ и ординат $\mathbb{Y}$, с последующим применением весов, которые также являются функциями позиции, $w(x, y)$. Тогда отклик простой клетки на изображение имеет вид

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x,y) I(x,y). \tag{9.15}$$

Точнее говоря, $w(x, y)$ – функция Габора:

$$w(x, y, \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp(-\beta_x x'^2 - \beta_y y'^2) \cos(fx' + \phi), \tag{9.16}$$

где

$$x' = (x - x_0)\cos(\tau) + (y - y_0)\sin(\tau) \tag{9.17}$$

и

$$y' = -(x - x_0)\sin(\tau) + (y - y_0)\cos(\tau). \tag{9.18}$$

Здесь $\alpha, \beta_x, \beta_y, f, \phi, x_0, y_0$ и τ – параметры, определяющие свойства функции Габора. На рис. 9.18 приведено несколько примеров функций Габора с разными значениями параметров.

Параметры x_0, y_0 и τ определяют систему координат. Для получения x' и y' из x и y производятся параллельный перенос и поворот. Точнее, простая клетка реагирует на признаки изображения с центром в точке (x_0, y_0) и на изменения яркости при движении вдоль прямой, повернутой на угол τ радиан относительно горизонтальной оси.

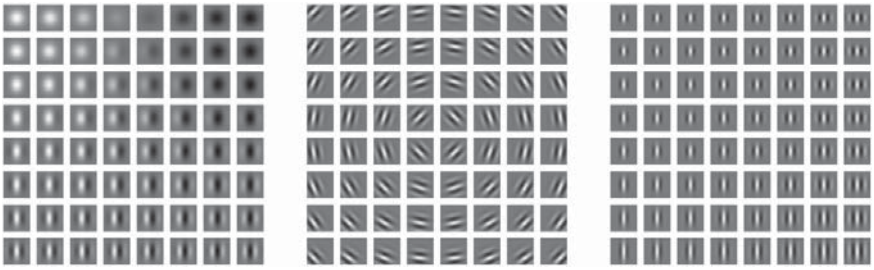

Рис. 9.18 ❖ Функция Габора с различными параметрами. Белым цветом обозначен большой положительный вес, черным – большой отрицательный вес, а серым фоном – нулевой вес. (*Слева*) Функции Габора с различными параметрами, описывающими систему координат: x_0, y_0 и τ. Каждой функции Габора в этой сетке соответствуют значения x_0 и y_0, пропорциональные ее позиции в сетке, а τ выбрано так, чтобы фильтр Габора бы чувствителен к направлению, исходящему из центра сетки. На двух остальных графиках x_0, y_0 и τ равны нулю. (*В центре*) Функции Габора с разными значениями параметров гауссианы β_x и β_y. Функции расположены в порядке возрастания ширины (убывания β_x) при движении по сетке слева направо и в порядке возрастания высоты (убывания β_y) при движении сверху вниз. На двух остальных графиках значения β фиксированы и равны ширине изображения, умноженной на 1.5. (*Справа*) Функции Габора с различными параметрами синусоиды f и ϕ. При движении по сетке сверху вниз возрастает f, а при движении слева направо возрастает ϕ. На двух остальных графиках $\phi = 0$, а f в 5 раз больше ширины изображения

Функция w, рассматриваемая как функция от x' и y', реагирует на изменение яркости вдоль оси x'. Она является произведением двух важных сомножителей: функции Гаусса и косинуса. Первый сомножитель $\alpha \exp(-\beta_x x'^2 - \beta_y y'^2)$ можно рассматривать как вентиль, гарантирующий, что простая клетка будет реагировать только на значения в окрестности точки с нулевыми координатами x' и y', иными словами, вблизи центра рецептивного поля клетки. Масштабный коэффициент α задает абсолютную величину отклика простой клетки, а параметры β_x и β_y определяют скорость спадания рецептивного поля.

Множитель $\cos(fx' + \phi)$ определяет реакцию простых клеток на изменение яркости вдоль оси x'. Параметр f – частота косинусоиды, а ϕ – сдвиг фазы.

Собирая все вместе, можно сказать, что это схематичное представление означает, что простая клетка реагирует на определенную пространственную частоту яркости в определенном направлении и в определенном месте. Возбуждение простой клетки максимально, если сигнал яркости в изображении имеет такую же фазу, как веса. Это происходит, когда изображение яркое в области положительности весов и темное – в области отрицательности. Торможение простых клеток максимально, когда сигнал яркости находится в противофазе с весами – изображение темное там, где веса положительны, и яркое – там, где веса отрицательны.

Схематичное представление сложной клетки состоит в том, что она вычисляет норму L^2 двумерного вектора, содержащего отклики двух простых клеток: $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. Важный частный случай – когда совпадают все параметры s_1 и s_0, кроме ϕ, а ϕ задан так, что сдвиг фазы s_1 и s_0 равен четверти периода. В этом случае s_0 и s_1 образуют **квадратурную пару**. Так определенная сложная клетка реагирует, когда взвешенное гауссово изображение $I(x, y)\exp(-\beta_x x'^2 - \beta_y y'^2)$ содержит высокоамплитудный синусоидальный сигнал с частотой f в направлении τ вблизи точки (x_0, y_0), независимо от сдвига фазы. Иными словами, сложная клетка инвариантна относительно небольших параллельных переносов изображения в направлении τ и инвертирования изображения (замены черного на белый и наоборот).

Самые поразительные аналогии между нейробиологией и машинным обучением связаны с визуальным сравнением признаков, найденных в результате обучения моделей, с теми, что обрабатываются зоной V1. В работе Olshausen and Field (1996) показано, что простой алгоритм обучения без учителя, разреженное кодирование, обучает признаки с рецептивными полями, напоминающими поля простых клеток. С тех пор было обнаружено, что самые разные алгоритмы статистического обучения обучают признаки с габороподобными функциями при применении к естественным изображениям. Сюда входит и большинство алгоритмов глубокого обучения, которые обучают эти признаки в первом слое. На рис. 9.19 показано несколько примеров. Поскольку существует так много разных алгоритмов обучения детекторов границ, трудно определенно утверждать, что некий конкретный алгоритм является «правильной» моделью мозга, исходя только из обучаемых им признаков (хотя, безусловно, можно считать плохим знаком, если алгоритм не обучает какой-то детектор границ при применении к естественным изображениям). Эти признаки – важная составная часть статистической структуры естественных изображений, и восстановить их позволяют разные подходы к статистическому моделированию. Обзор состояния дел в области статистики естественных изображений имеется в работе Hyvärinen et al. (2009).

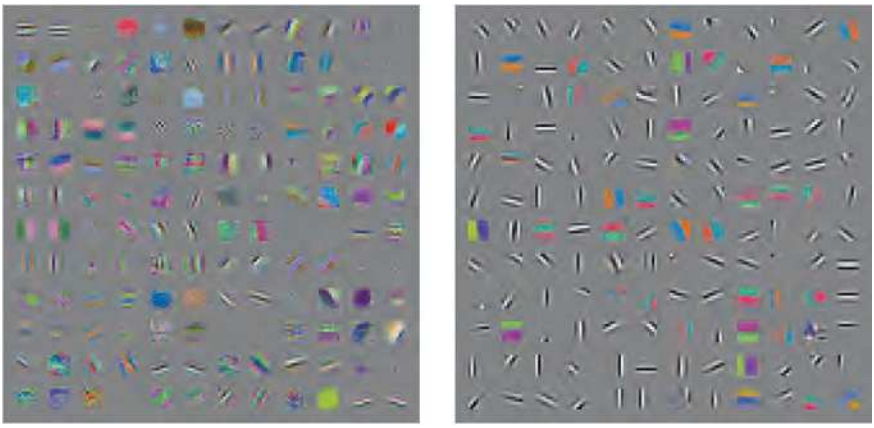

Рис. 9.19 ❖ Многие алгоритмы машинного обучения обучают признаки для обнаружения любых границ или границ определенного цвета в естественных изображениях. Такие детекторы напоминают функции Габора, принимающие участие в работе первичной зрительной коры. (*Слева*) Веса, обученные алгоритмом обучения без учителя (разреженное кодирование типа Spike-and-Slab) в применении к небольшим участкам изображения. (*Справа*) Сверточные ядра, обученные с учителем первым слоем сверточной maxout-сети. Соседние пары фильтров управляют одним и тем же maxout-блоком

9.11. Сверточные сети и история глубокого обучения

Сверточные сети сыграли важную роль в истории глубокого обучения. Это яркий пример успешного применения идей, высказанных в процессе изучения человеческого мозга, к машинному обучению. Заодно они оказались одними из первых глубоких моделей хорошего качества, задолго до того как глубокие модели вообще были признаны жизнеспособными. Сверточные сети были также одними из первых нейронных сетей, нашедших применение в важных коммерческих приложениях, и до сих пор они остаются в первых рядах коммерческих применений глубокого обучения. Например, в 1990-е годы исследовательская группа по нейронным сетям в компании AT&T разработала сверточную сеть для распознавания чеков (LeCun et al., 1998b). К концу 1990-х годов эта система была развернута компанией NEC и распознавала 10% всех чеков в США. Позже несколько систем распознавания символов и рукописных текстов на основе сверточных сетей было развернуты корпорацией Microsoft (Simard et al., 2003). Подробнее об этих и других современных применениях сверточных сетей см. главу 12. Более полный обзор истории сверточных сетей до 2010 года см. в работе LeCun et al. (2010).

Системы на основе сверточных сетей выигрывали много конкурсов. Современный всплеск коммерческого интереса к глубокому обучению начался, когда система, описанная в работе Krizhevsky et al. (2012), победила в конкурсе по распознаванию объектов ImageNet, но сверточные сети побеждали и в других соревнованиях по машинному обучению и компьютерному зрению задолго до того, хотя это и не вызывало такого ажиотажа.

Сверточные сети были одними из первых работающих глубоких сетей, обученных с применением обратного распространения. Не вполне понятно, почему сверточные сети добились успеха там, где сети общего назначения с обратным распространением потерпели неудачу. Может быть, дело в том, что сверточные сети вычислительно эффективнее полносвязных сетей, поэтому было проще ставить эксперименты и оптимизировать реализацию и гиперпараметры. Кроме того, чем больше сеть, тем легче ее обучить. На современном оборудовании большие полносвязные сети дают неплохие результаты на многих задачах даже при использовании наборов данных и функций активации, которые были доступны и популярны в те времена, когда считалось, что от полносвязных сетей невозможно добиться хорошего качества. Возможно также, что основные барьеры на пути к успеху нейронных сетей были чисто психологического свойства (специалисты-практики не ожидали от них хорошего качества, а потому не особенно старались их использовать). Как бы то ни было, нам повезло, что сверточные сети хорошо работали несколько десятков лет тому назад. Во многих смыслах они передали эстафету остальным разделам глубокого обучения и проложили путь принятию нейронных сетей в целом.

Сверточные сети позволяют специализировать нейронные сети для работы с данными, имеющими четко выраженную сеточную топологию, и хорошо масштабировать такие модели к задачам очень большого размера. Особенно успешным этот подход оказался в применении к двумерным изображениям. Для обработки одномерных последовательных данных следует обратиться к еще одной эффективной специализации: рекуррентным нейронным сетям. Ими мы и займемся в следующей главе.

Глава 10

Моделирование последовательностей: рекуррентные и рекурсивные сети

Рекуррентные нейронные сети, или РНС (Rumelhart et al., 1986a), – это семейство нейронных сетей для обработки последовательных данных. Если сверточная сеть предназначена для обработки сетки значений $\mathbf{X}$ типа изображения, то рекуррентная нейронная сеть предназначена для обработки последовательности значений $\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}$. Если сверточная сеть легко масштабируется на изображения большой ширины и высоты, а некоторые сети даже могут обрабатывать изображения переменного размера, то рекуррентная сеть масштабируется на гораздо более длинные последовательности, чем было бы практически возможно для неспециализированных нейронных сетей. Большинство рекуррентных сетей способно также обрабатывать последовательности переменной длины.

Для перехода от многослойных сетей к рекуррентным мы воспользуемся одной из ранних идей машинного обучения и статистического моделирования, появившейся еще в 1980-е годы: разделение параметров между различными частями модели. Разделение параметров позволяет применить модель к примерам различной формы (в данном случае – длины) и выполнить для них обобщение. Если бы для каждого временного индекса были отдельные параметры, то мы не смогли бы ни обобщить модель на длины последовательностей, не встречавшиеся на этапе обучения, ни распространить статистическую силу на последовательности разной длины и на разные моменты времени. Такое разделение особенно важно, если некоторая часть информации может встречаться в нескольких местах последовательности. Например, рассмотрим два предложения: «Я ездил в Непал в 2009 году» и «В 2009 году я ездил в Непал». Когда мы просим модель прочитать каждое предложение и выделить год, в котором рассказчик ездил в Непал, мы ожидаем получить 2009 вне зависимости от того, находится интересующая нас информация в шестом или во втором слове. Предположим, что мы обучили сеть прямого распространения обрабатывать предложения фиксированной длины. В традиционной полносвязной сети были бы отдельные параметры для каждого входного признака, поэтому потребовалось бы обучать всем

правилам языка отдельно в каждой позиции в предложении. А в рекуррентной нейронной сети одни и те же веса разделяются между несколькими временными шагами.

Родственная идея – применить свертку к одномерной временной последовательности. Такой сверточный подход лежит в основе нейронных сетей с временной задержкой (Lang and Hinton, 1988; Waibel et al., 1989; Lang et al., 1990). Операция свертки позволяет сети разделять параметры во времени, но является «мелкой». На выходе свертки получается последовательность, каждый член которой – функция от небольшого числа соседних членов входной последовательности. Идея разделения параметров проявляется в применении одного и того же ядра свертки на каждом временном шаге. В рекуррентных сетях разделение параметров происходит по-другому. Каждый выходной член – функция предыдущих выходных членов и порождается с помощью применения одного и того же правила обновления к предыдущим членам. Такая рекуррентная формулировка дает возможность разделять параметры в очень глубоком графе вычислений.

Для простоты изложения будем считать, что РНС воздействует на последовательность векторов $\boldsymbol{x}^{(t)}$ с индексом временного шага t в диапазоне от 1 до τ. На практике рекуррентные сети обычно применяются к мини-пакетам таких последовательностей с разной длиной последовательности τ для каждого элемента мини-пакета. Для простоты обозначений мы опускаем индексы мини-пакетов. Кроме того, индекс временного шага необязательно буквально соответствует течению времени в реальном мире. Иногда это просто позиция внутри последовательности. РНС можно также применять к двумерным пространственным данным типа изображений, а если речь идет о данных, в которых участвует время, то в сети могут существовать связи, ведущие назад во времени, при условии что вся последовательность известна до передачи ее сети.

В этой главе мы обобщим идею графа вычислений, включив в него циклы. Циклы представляют влияние текущего значения переменной на ее же значение на будущем временном шаге. С помощью таких графов вычислений можно определять рекуррентные нейронные сети. Затем мы опишем различные способы построения, обучения и использования рекуррентных нейронных сетей.

За дополнительными сведениями о рекуррентных нейронных сетях отсылаем читателя к книге Graves (2012).

10.1. Развертка графа вычислений

Граф вычислений – это формальный способ описать структуру множества вычислений, например необходимых для отображения входов и параметров на выходы и потерю. Общее введение в эту тему приведено в разделе 6.5.1. А сейчас мы объясним идею **развертки** (unfolding) рекурсивного или рекуррентного вычисления в граф вычислений с повторяющейся структурой, обычно соответствующей цепочке событий. Развертка такого графа приводит к разделению параметров между структурными элементами глубокой сети.

Например, рассмотрим классическую форму динамической системы:

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}), \tag{10.1}$$

где $\boldsymbol{s}^{(t)}$ называется состоянием системы.

Выражение (10.1) рекуррентное, потому что определение $\boldsymbol{s}$ в момент t ссылается на то же самое определение в момент $t - 1$.

Для конечного числа временных шагов τ граф можно развернуть, применив это определение $\tau - 1$ раз. Например, если развернуть выражение (10.1) для $\tau = 3$ шагов, то получим:

$$\boldsymbol{s}^{(3)} = f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) \tag{10.2}$$

$$= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}). \tag{10.3}$$

После такой развертки путем повторного применения определения получается выражение, не содержащее рекурсии. Такое выражение можно представить традиционным ациклическим ориентированным графом вычислений. Развернутый граф вычислений выражений (10.1) и (10.3) показан на рис. 10.1.

Рис. 10.1 ❖ Классическая динамическая система, описываемая выражением (10.1), в виде развернутого графа вычислений. Каждая вершина представляет состояние в некоторый момент t, а функция f отображает состояние в момент t на состояние в момент $t + 1$. Одни и те же параметры (значение $\boldsymbol{\theta}$, параметризующее f) используются на всех временных шагах

В качестве другого примера рассмотрим динамическую систему, управляемую внешним сигналом $\boldsymbol{x}(t)$:

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}). \tag{10.4}$$

Теперь состояние содержит информацию обо всей прошлой последовательности.

Рекуррентные нейронные сети можно строить разными способами. Как почти любую функцию можно рассматривать как нейронную сеть прямого распространения, так и практически любую рекуррентную функцию можно рассматривать как рекуррентную нейронную сеть. Во многих рекуррентных нейронных сетях используется уравнение (10.5) или аналогичное для задания значений скрытых блоков. Чтобы подчеркнуть, что состояние – это на самом деле скрытые блоки сети, перепишем уравнение (10.4), используя для представления состояния переменную $\boldsymbol{h}$:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}). \tag{10.5}$$

Такая сеть показана на рис. 10.2; в типичных РНС есть дополнительные архитектурные особенности, например выходные слои, которые читают информацию из состояния $\boldsymbol{h}$, чтобы сделать предсказание.

Когда рекуррентную сеть обучают решать задачу, в которой требуется предсказывать будущее по прошлому, сеть обычно обучается использовать $\boldsymbol{h}^{(t)}$ как сводку относящихся к задаче аспектов последовательности входных данных, предшествующей моменту t. В общем случае в сводке по необходимости утрачивается часть информации, потому что она отображает последовательность произвольной длины ($\boldsymbol{x}^{(t)}$, $\boldsymbol{x}^{(t-1)}$, $\boldsymbol{x}^{(t-2)}$, ..., $\boldsymbol{x}^{(2)}$, $\boldsymbol{x}^{(1)}$) на вектор фиксированной длины $\boldsymbol{h}^{(t)}$. В зависимости от критерия обучения некоторые аспекты прошлой последовательности могут запоминаться в сводке с большей точностью, чем остальные. Например, если РНС используется для статистического моделирования языка, как правило, для предсказания следующего слова по известным предыдущим, то достаточно сохранить только информацию, не-

обходимую для предсказания остатка предложения. Самая трудная ситуация складывается, когда мы хотим, чтобы вектор $\boldsymbol{h}^{(t)}$ был достаточно полным для приближенного восстановления входной последовательности, как в автокодировщиках (глава 14).

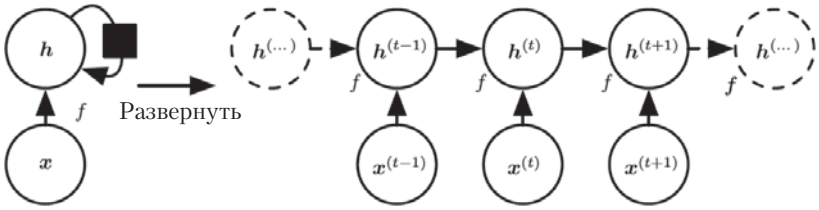

Рис. 10.2 ❖ Рекуррентная сеть без выходов. Эта сеть просто обрабатывает информацию из входа $\boldsymbol{x}$, включая ее в состояние $\boldsymbol{h}$, которое передается дальше во времени. (*Слева*) Принципиальная схема. Черный квадратик обозначает задержку на один временной шаг. (*Справа*) Та же сеть в виде развернутого графа вычислений, в котором каждая вершина ассоциирована с одним моментом времени

Уравнение (10.5) можно изобразить двумя способами. Первый способ – нарисовать диаграмму, содержащую по одному узлу для каждой компоненты, которая могла бы существовать в физической реализации модели, например в биологической нейронной сети. В этом случае сеть определяет схему, которая содержит физические детали и работает в режиме реального времени, а ее текущее состояние может оказывать влияние на будущее; этот вариант изображен в левой части рис. 10.2. В этой главе черный квадратик на принципиальной схеме сети означает, что взаимодействие имеет место с задержкой на один временной шаг, т. е. происходит переход из состояния в момент t в состояние в момент $t + 1$. Другой способ изобразить РНС – нарисовать развернутый граф вычислений, в котором каждая компонента представлена многими переменными состояния, по одной на временной шаг. Каждая переменная на каждом временном шаге изображается в виде отдельной вершины графа вычислений, как в правой части рис. 10.2. Разверткой мы называем операцию, которая отображает принципиальную схему в граф вычислений с повторяющимися частями. Размер развернутого графа зависит от длины последовательности.

Мы можем представить развернутое рекуррентное выражение после t шагов функцией $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, ..., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \tag{10.6}$$

$$= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}). \tag{10.7}$$

Функция $g^{(t)}$ принимает на входе всю прошлую последовательность $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, ..., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ и порождает текущее состояние, но развернутая рекуррентная структура позволяет представить $g^{(t)}$ в виде многократного применения функции f. Таким образом, процесс развертки дает два важных преимущества:

1) независимо от длины последовательности размер входа обученной модели всегда один и тот же, поскольку он описывается в терминах перехода из одного состояния в другое, а не в терминах истории состояний переменной длины;
2) одну и ту же функцию перехода f с одними и теми же параметрами можно использовать на каждом шаге.

Эти два фактора позволяют обучить одну модель f, которая действует на всех временных шагах и для последовательностей любой длины, не прибегая к обучению отдельных моделей $g^{(t)}$ для каждого временного шага. Обучение единственной разделяемой модели открывает возможность обобщения на такие длины последовательности, которые не встречались в обучающем наборе, и позволяет оценивать модель при наличии гораздо меньшего числа обучающих примеров, чем понадобилось бы без разделения параметров.

Как у рекуррентного, так и у развернутого графа есть свои достоинства. Рекуррентный граф лаконичен, развернутый дает явное описание всех вычислений. Кроме того, развернутый граф иллюстрирует идею протекания информации во времени вперед (вычисление выходов и потерь) и назад (вычисление градиентов), поскольку явно содержит путь, по которому течет информация.

10.2. Рекуррентные нейронные сети

Вооружившись механизмами развертки графов и разделения параметров, мы можем перейти к проектированию разнообразных рекуррентных нейронных сетей.

Вот несколько важных паттернов проектирования таких сетей:

- рекуррентные сети, порождающие выход на каждом временном шаге и имеющие рекуррентные связи между скрытыми блоками (рис. 10.3);
- рекуррентные сети, порождающие выход на каждом временном шаге и имеющие рекуррентные связи только между выходами на одном временном шаге и скрытыми блоками на следующем (рис. 10.4);
- рекуррентные сети с рекуррентными связями между скрытыми блоками, которые читают последовательность целиком, а затем порождают единственный выход (рис. 10.5).

На рис. 10.3 изображен достаточно репрезентативный пример, к которому мы не раз будем возвращаться в этой главе.

Рекуррентная нейронная сеть на рис. 10.3 и уравнение (10.8) универсальны в том смысле, что любая функция, вычислимая машиной Тьюринга, может быть вычислена и такой рекуррентной сетью конечного размера. Результат можно прочитать из РНС после выполнения числа шагов, асимптотически линейно зависящего от числа временных шагов машины Тьюринга и от длины входной последовательности (Siegelmann and Sontag, 1991; Siegelmann, 1995; Siegelmann and Sontag, 1995; Hyotyniemi, 1996). Функции, вычислимые машиной Тьюринга, дискретны, и потому эти результаты относятся к точной реализации функции, а не к аппроксимациям. Когда РНС используется как машина Тьюринга, она принимает на входе двоичную последовательность, а ее выходы можно дискретизировать для получения двоичного результата. В таких предположениях можно вычислить любую функцию с помощью одной конкретной РНС конечного размера (в работе Siegelmann and Sontag [1995] используется РНС с 886 блоками). «Входом» машины Тьюринга является спецификация вычисляемой функции, поэтому той же сети, которая моделирует машину Тьюринга, достаточно для решения всех задач. Теоретическая РНС, применяемая в доказательстве, умеет моделировать неограниченный стек, представляя его активации и веса рациональными числами неограниченной точности.

Теперь выведем уравнения прямого распространения для РНС, изображенной на рис. 10.3. На этом рисунке не показана конкретная функция активации для скрытых

блоков. Будем предполагать, что это гиперболический тангенс. Кроме того, на рисунке точно не указана форма выхода и функции потерь. Будем предполагать, что выход дискретный, как если бы РНС применялась для предсказания слов или символов. Естественный способ представления дискретных величин – рассматривать выход как ненормированные логарифмические вероятности каждого возможного значения дискретной величины. Тогда на этапе постобработки можно применить операцию softmax и получить вектор $\hat{\boldsymbol{y}}$ нормированных вероятностей. Прямое распространение начинается с задания начального состояния $\boldsymbol{h}^{(0)}$. Затем для каждого временного шага от $t = 1$ до $t = \tau$ применяем следующие уравнения обновления:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \tag{10.8}$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}) \tag{10.9}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \tag{10.10}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}), \tag{10.11}$$

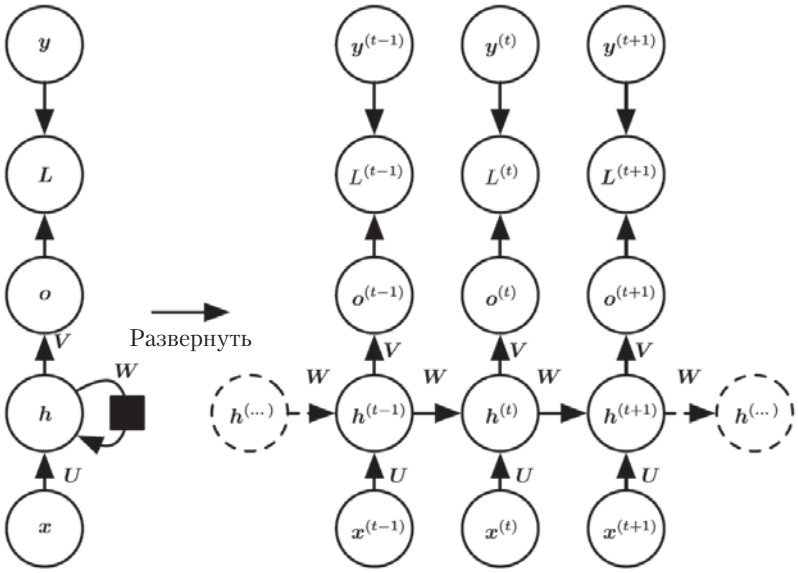

Рис. 10.3 ❖ Граф вычислений потерь при обучении рекуррентной сети, которая отображает входную последовательность значений $\boldsymbol{x}$ в соответствующую выходную последовательность значений $\boldsymbol{o}$. Функция потерь L измеряет, насколько далеко каждый элемент $\boldsymbol{o}$ отстоит от соответствующей метки $\boldsymbol{y}$. В случае применения к выходам функции softmax можно предполагать, что $\boldsymbol{o}$ – ненормированные логарифмические вероятности. Внутри функция L вычисляет $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ и сравнивает эту величину с меткой $\boldsymbol{y}$. В РНС имеются связи между входным и скрытым слоями, параметризованные матрицей весов $\boldsymbol{U}$, рекуррентные связи между скрытыми блоками, параметризованные матрицей весов $\boldsymbol{W}$, и связи между скрытым и выходным слоями, параметризованные матрицей весов $\boldsymbol{V}$. Уравнение (10.8) определяет прямое распространение в этой модели. (*Слева*) РНС и ее функция потерь, представленные в виде рекуррентных связей. (*Справа*) То же в виде развернутого во времени графа вычислений, в котором каждая вершина ассоциирована с одним моментом времени

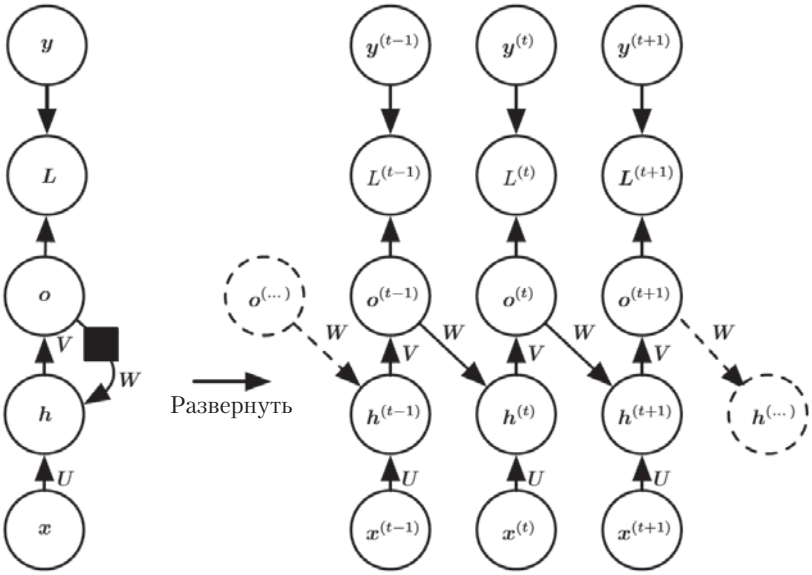

Рис. 10.4 ❖ РНС, в которой единственным видом рекурсии является обратная связь между выходным и скрытым слоями. На каждом временном шаге t вход обозначен $\boldsymbol{x}_t$, активации скрытого слоя – $\boldsymbol{h}^{(t)}$, выходы – $\boldsymbol{o}^{(t)}$, метки – $\boldsymbol{y}^{(t)}$, а потеря – $L^{(t)}$. (Слева) Принципиальная схема. (Справа) Развернутый граф вычислений. Такая РНС менее мощная (способна выразить меньшее множество функций), чем РНС из семейства на рис. 10.3. РНС, показанная на рис. 10.3, может поместить любую информацию о прошлом в скрытое представление $\boldsymbol{h}$ и передать $\boldsymbol{h}$ в будущее. РНС на этом рисунке обучена помещать конкретное выходное значение в $\boldsymbol{o}$, и $\boldsymbol{o}$ – единственная информация, которую разрешено передавать в будущее. Не существует прямых горизонтальных связей, исходящих из $\boldsymbol{h}$. Предыдущий $\boldsymbol{h}$ связан с настоящим только косвенно – с помощью порожденных им предсказаний. Если размерность вектора $\boldsymbol{o}$ не слишком велика, то в нем обычно отсутствует какая-то важная информация о прошлом. Поэтому такая РНС менее мощная, но зато ее легче обучить, потому что каждый временной шаг можно обучать отдельно от всех остальных, благодаря чему возможно в большей степени распараллелить обучение (см. раздел 10.2.1)

где параметрами являются векторы смещения $\boldsymbol{b}$ и $\boldsymbol{c}$, а также матрицы весов $\boldsymbol{U}$, $\boldsymbol{V}$ и $\boldsymbol{W}$ соответственно для связей между входным и скрытым слоями, между скрытым и выходным слоями и между скрытыми блоками. Это пример рекуррентной сети, которая отображает входную последовательность на выходную той же длины. Полная потеря для данной входной последовательности $\boldsymbol{x}$ в совокупности с последовательностью меток $\boldsymbol{y}$ равна сумме потерь по всем временным шагам. Например, если $L^{(t)}$ – отрицательное логарифмическое правдоподобие $y^{(t)}$ при условии $\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(t)}$, то

$$L(\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(\tau)}\}) \tag{10.12}$$

$$= \sum_t L^{(t)} \tag{10.13}$$

$$= -\sum_t \log p_{\text{model}}(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}), \tag{10.14}$$

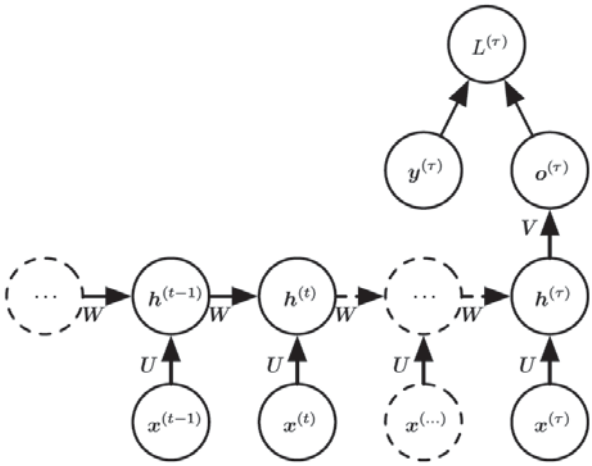

Рис. 10.5 ❖ Развернутая во времени рекуррентная нейронная сеть с единственным выходом в конце последовательности. Такую сеть можно использовать для агрегирования последовательности и порождения представления фиксированного размера, подаваемого на вход следующего этапа обработки. В самом конце может быть сравнение с меткой (как показано на рисунке), но можно также получить градиент $\boldsymbol{o}^{(t)}$ посредством обратного распространения от последующих модулей

где $p_{\text{model}}(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(t)}\})$ берется из элемента $y^{(t)}$ выходного вектора модели $\hat{\boldsymbol{y}}^{(t)}$. Вычисление градиента этой функции потерь относительно параметров – дорогая операция. Для вычисления градиента необходимо выполнить прямое распространение, двигаясь слева направо по развернутому графу на рис. 10.3, а затем обратное распространение, двигаясь справа налево по тому же графу. Время работы имеет порядок $O(\tau)$, и его нельзя уменьшить за счет распараллеливания, потому что граф прямого распространения принципиально последовательный: каждый временной шаг можно обсчитать только после завершения предыдущего. Состояния, вычисленные во время прямого прохода, необходимо хранить до повторного использования на обратном проходе, поэтому объем потребляемой памяти также имеет порядок $O(\tau)$. Алгоритм обратного распространения, применяемый к развернутому графу и имеющий сложность $O(\tau)$, называется **обратным распространением во времени** (back-propagation through time – BPTT) и будет подробно рассмотрен в разделе 10.2.2. Таким образом, сеть с рекурсией между скрытыми блоками является очень мощной, но дорогой для обучения. Существует ли альтернатива?

10.2.1. Форсирование учителя и сети с рекурсией на выходе

Сеть, в которой имеются только рекуррентные связи между выходными блоками на одном временном шаге и скрытыми блоками на следующем (рис. 10.4), является строго менее мощной, потому что ей недостает рекуррентных связей между скрытыми блоками. Так, с ее помощью нельзя смоделировать универсальную машину Тьюринга. Поскольку в сети нет рекуррентных связей между скрытыми блоками, необходимо, чтобы выходные блоки запоминали всю ту информацию о прошлом, которую сеть будет использовать для предсказания будущего. Поскольку выходные блоки

явно обучаются совпадению с метками обучающего набора, то маловероятно, что они запомнят нужную информацию о прошлой истории входов, – разве что пользователь знает, как описать полное состояние системы, и включает эти сведения в состав выходных меток обучающего набора. Но у исключения рекуррентных связей между скрытыми блоками есть и преимущество – для любой функции потерь, основанной на сравнении предсказания в момент t с меткой в момент t, все временные шаги полностью независимы. Следовательно, обучение можно распараллелить и вычислять градиент для каждого шага t изолированно. Нет нужды вычислять сначала выход для предыдущего временного шага, потому что обучающий набор уже содержит идеальное значение этого выхода.

Модели, в которых имеются рекуррентные связи, идущие от выходов обратно в модель, можно обучить методом **форсирования учителя** (teacher forcing). Форсирование учителя – процедура, берущая начало в критерии максимального правдоподобия, когда во время обучения модель получает истинную метку $y^{(t)}$ в качестве входа в момент $t + 1$. Как это происходит, можно видеть, рассмотрев последовательность с двумя временными шагами. Критерий условного максимального правдоподобия имеет вид:

$$\log p(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) \tag{10.15}$$

$$= \log p(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) + \log p(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}). \tag{10.16}$$

Здесь мы видим, что в момент $t = 2$ модель обучается максимизировать условную вероятность $\boldsymbol{y}^{(2)}$ при известной до этого момента последовательности $\boldsymbol{x}$ и предыдущем значении $\boldsymbol{y}$ из обучающего набора. Таким образом, критерий максимального правдоподобия говорит, что на этапе обучения мы должны подать обратно на вход не собственный выход модели, а значения меток, описывающие, каким должен быть правильный выход. Это показано на рис. 10.6.

Исходным обоснованием форсирования учителем было желание избежать обратного распространения во времени в моделях, где нет связей между скрытыми блоками. Его можно применять и к моделям, где такие связи есть, при условии что имеются связи между выходом на одном временном шаге и значениями, вычисленными на следующем шаге. Но коль скоро скрытые блоки становятся функцией предшествующих временных шагов, необходим алгоритм BPTT. Поэтому некоторые модели можно обучать, применяя и форсирование учителя, и BPTT.

Недостаток строгого метода форсирования учителя проявляется, если впоследствии предполагается использовать сеть в режиме **с разомкнутой обратной связью**, когда выходы сети (или примеры, выбранные из выходного распределения) подаются обратно на вход. В таком случае входные данные, которые сеть видит в процессе обучения, могут сильно отличаться от того, что она видит на этапе тестирования. Один из способов сгладить эту проблему – обучать как на форсированных, так и на свободных входах, например выдавая в качестве предсказания правильную метку через несколько шагов в будущем с использованием развернутых рекуррентных путей между выходом и входом. Таким образом, можно обучить сеть принимать во внимание входные условия (скажем, те, что она генерирует сама в свободном режиме), которые не встречались во время обучения, и отображать состояние обратно в такое, которое заставит сеть генерировать правильные выходы после нескольких шагов. Еще один подход (Bengio et al., 2015b) сократить разрыв между входами, предъявляемыми на этапе обучения и на этапе тестирования, – случайно выбирать между подачей на вход

сгенерированных или фактических значений. В этом подходе используется стратегия обучения по плану, с тем чтобы постепенно увеличивать долю сгенерированных значений на входе.

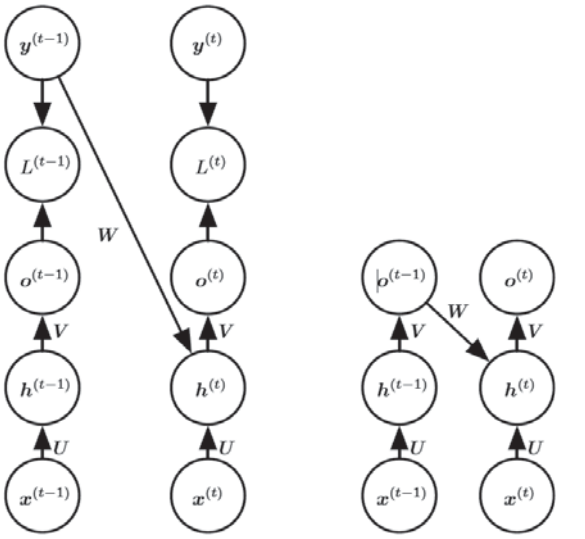

Рис. 10.6 ❖ Форсирование учителя – метод обучения, применимый к РНС, в которых есть связи между выходом и скрытым состоянием на следующем временном шаге. (*Слева*) На этапе обучения мы подаем *истинный выход* $\boldsymbol{y}^{(t)}$, взятый из обучающего набора, на вход $\boldsymbol{h}^{(t+1)}$. (*Справа*) После того как модель развернута, истинный выход, вообще говоря, неизвестен. В таком случае мы аппроксимируем истинный выход $\boldsymbol{y}^{(t)}$ выходом модели $\boldsymbol{o}^{(t)}$ и подаем этот выход обратно в модель

10.2.2. Вычисление градиента в рекуррентной нейронной сети

Вычисление градиента в рекуррентной нейронной сети не вызывает трудностей. Нужно просто применить обобщенный алгоритм обратного распространения из раздела 6.5.6 к развернутому графу вычислений. Никаких специальных алгоритмов не нужно. Градиенты, полученные в результате обратного распространения, можно затем использовать в сочетании с любым универсальным градиентным методом для обучения РНС.

Чтобы составить интуитивное представление о поведении алгоритма BPTT, приведем пример вычисления градиентов для уравнений РНС выше (уравнение 10.8 и 10.12). В нашем графе вычислений имеются параметры $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$ и $\boldsymbol{c}$, а также последовательность вершин, индексированных временем t: $\boldsymbol{x}^{(t)}$, $\boldsymbol{h}^{(t)}$, $\boldsymbol{o}^{(t)}$ и $L^{(t)}$. Для каждой вершины $\mathbf{N}$ мы должны рекурсивно вычислить градиент $\nabla_{\mathbf{N}} L$, зная градиенты, вычисленные в вершинах, следующих за ней в графе. Рекурсия начинается с вершин, непосредственно предшествующих окончательной потере:

$$\frac{\partial L}{\partial L^{(t)}} = 1. \tag{10.17}$$

Предполагается, что выходы $\boldsymbol{o}^{(t)}$ используются в качестве аргумента функции softmax для получения вектора вероятностей выходов $\hat{\boldsymbol{y}}$. Предполагается также, что функция потерь – это отрицательное логарифмическое правдоподобие истинной метки $y^{(t)}$ при известных к этому моменту входах. Градиент $\nabla_{\boldsymbol{o}^{(t)}} L$ по выходам в момент t для всех i, t имеет вид:

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i, y^{(t)}}. \tag{10.18}$$

Мы движемся в направлении от конца последовательности к началу. В последний момент времени τ у $\boldsymbol{h}^{(\tau)}$ есть только один потомок $\boldsymbol{o}^{(\tau)}$, поэтому вычислить градиент просто:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L. \tag{10.19}$$

Затем можно совершать итерации назад во времени для обратного распространения градиентов от $t = \tau - 1$ до $t = 1$. Заметим, что потомками $\boldsymbol{h}^{(t)}$ (для $t < \tau$) являются $\boldsymbol{o}^{(t)}$ и $\boldsymbol{h}^{(t+1)}$. Следовательно, градиент равен

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \right)^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \right)^\top (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.20}$$

$$= \boldsymbol{W}^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}(1 - (\boldsymbol{h}^{(t+1)})^2) + \boldsymbol{V}^\top (\nabla_{\boldsymbol{o}^{(t)}} L), \tag{10.21}$$

где $\operatorname{diag}(1 - (\boldsymbol{h}^{(t+1)})^2)$ – диагональная матрица с элементами $1 - (h_i^{(t+1)})^2$. Это якобиан функции гиперболического тангенса, ассоциированный со скрытым блоком i в момент $t + 1$.

Получив градиенты по внутренним вершинам графа вычислений, мы можем затем получить градиенты по вершинам параметров. Поскольку параметры разделяются между временными шагами, следует аккуратно подходить к обозначениям аналитических операций с участием этих переменных. В нужных нам уравнениях используется метод `bprop` из раздела 6.5.6, который вычисляет вклад в градиент одного ребра графа вычислений. Но оператор $\nabla_{\boldsymbol{W}} f$, применяемый в математическом анализе, принимает во внимание вклад $\boldsymbol{W}$ в значение f, вносимый *всеми* ребрами графа вычислений. Для разрешения этой неоднозначности мы введем фиктивные переменные $\boldsymbol{W}^{(t)}$, определенные как копии $\boldsymbol{W}$, только каждая $\boldsymbol{W}^{(t)}$ используется лишь на временном шаге t. Тогда $\nabla_{\boldsymbol{W}^{(t)}} f$ можно использовать для обозначения вклада весов на шаге t в градиент. В этой нотации градиенты по остальным параметрам имеют вид

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \right)^\top \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L, \tag{10.22}$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}} \right)^\top \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \nabla_{\boldsymbol{h}^{(t)}} L, \tag{10.23}$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_t \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t (\nabla_{\boldsymbol{o}^{(t)}} L) \boldsymbol{h}^{(t)\top}, \tag{10.24}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_t \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \tag{10.25}$$

$$= \sum_t \text{diag}(1-(\boldsymbol{h}^{(t)})^2)(\nabla_{\boldsymbol{h}^{(t)}} L)\boldsymbol{h}^{(t-1)^\top}, \tag{10.26}$$

$$\nabla_U L = \sum_t \sum_t \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{U^{(t)}} h_i^{(t)} \tag{10.27}$$

$$= \sum_t \text{diag}(1-(\boldsymbol{h}^{(t)})^2)(\nabla_{\boldsymbol{h}^{(t)}} L)\boldsymbol{x}^{(t)^\top}. \tag{10.28}$$

Нам не нужно вычислять градиент по $\boldsymbol{x}^{(t)}$ для обучения, потому что среди его предков в графе вычислений, определяющем потерю, нет параметров.

10.2.3. Рекуррентные сети как ориентированные графические модели

В разработанном выше примере рекуррентной сети в роли потери $L^{(t)}$ выступала перекрестная энтропия между метками $\boldsymbol{y}^{(t)}$ и выходами $\boldsymbol{o}^{(t)}$. Как и в сетях прямого распространения, в рекуррентной сети, в принципе, можно использовать почти любую функцию потерь. Выбирать ее следует в зависимости от задачи. Как и в сети прямого распространения, мы обычно хотим интерпретировать выход РНС как распределение вероятности, а для определения потерь используем ассоциированную с этим распределением перекрестную энтропию. Например, среднеквадратическая ошибка – это потеря в виде перекрестной энтропии, ассоциированной с нормальным распределением выхода с нулевым средним и единичной дисперсией, – как и в сети прямого распространения.

Когда в качестве целевой функции обучения используется логарифмическое правдоподобие, как в уравнении (10.12), мы обучаем РНС оценивать условное распределение следующего элемента последовательности $\boldsymbol{y}^{(t)}$ при условии прошлых входов:

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(t)}) \tag{10.29}$$

или если модель включает связи между выходом на одном временном шаге и на следующем за ним, то:

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(t-1)}). \tag{10.30}$$

Разложение совместного распределения последовательности значений $\boldsymbol{y}$ в серию одношаговых предсказаний вероятности – это один из способов запомнить полное совместное распределение во всей последовательности. Если мы подаем прошлые значения $\boldsymbol{y}$ в качестве входов, обусловливающих предсказание на следующем шаге, то в ориентированной графической модели не будет ребер, ведущих из любого $\boldsymbol{y}^{(i)}$ в прошлом в текущий $\boldsymbol{y}^{(t)}$. В этом случае выходы $\boldsymbol{y}$ условно независимы при условии последовательности значений $\boldsymbol{x}$. Если же мы подаем фактические значения $\boldsymbol{y}$ (не предсказания, а наблюдаемые или сгенерированные значения) обратно в сеть, то ориентированная графическая модель содержит ребра из всех значений $\boldsymbol{y}^{(i)}$ в прошлом в текущее значение $y^{(t)}$.

В качестве простого примера рассмотрим случай, когда РНС моделирует только последовательность скалярных случайных величин $\mathbb{Y} = \{y^{(1)}, ..., y^{(\tau)}\}$ без дополнительных входов x. Входом на временном шаге t является выход шага $t - 1$. Тогда РНС определяет ориентированную модель над переменными y. Параметризуем совместное распределение этих наблюдений, применив цепное правило условных вероятностей (формула 3.6):

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \ldots, \mathbf{y}^{(1)}), \tag{10.31}$$

где при $t = 1$ справа от вертикальной черты, конечно, ничего нет. При такой модели отрицательное логарифмическое правдоподобие множества значений $\{y^{(1)}, ..., y^{(\tau)}\}$ имеет вид

$$L = \sum_{t} L^{(t)}, \tag{10.32}$$

где

$$L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, ..., y^{(1)}). \tag{10.33}$$

Ребра графической модели показывают, какие переменные прямо зависят от других переменных. Многие графические модели стремятся повысить статистическую и вычислительную эффективность, опуская ребра, которым не соответствуют сильные взаимодействия. Например, часто принимают марковское предположение о том, что графическая модель должна содержать только ребра, ведущие из $\{\mathrm{y}^{(t-k)}, ..., \mathrm{y}^{(t-1)}\}$ в $\mathrm{y}^{(t)}$, а не ребра, описывающие всю историю. Но в некоторых случаях мы полагаем, что на следующий элемент последовательности должны оказывать влияние все прошлые входы. РНС полезны, когда мы считаем, что распределение $\mathrm{y}^{(t)}$ может зависеть от значения $\mathrm{y}^{(i)}$ из отдаленного прошлого способом, не отраженным во влиянии $\mathrm{y}^{(i)}$ на $\mathrm{y}^{(t-1)}$.

Интерпретировать РНС как графическую модель можно, в частности, рассматривая сеть как определение графической модели, имеющей вид полного графа, способного представить прямые зависимости между любой парой значений y. Модель с такой структурой показана на рис. 10.7. Интерпретация РНС как полного графа основана на игнорировании скрытых блоков $\boldsymbol{h}^{(t)}$, которые исключаются из модели.

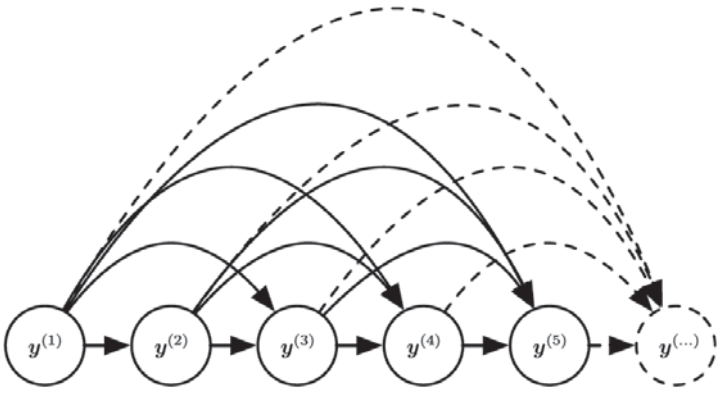

Рис. 10.7 ❖ Полносвязная графическая модель последовательности $y^{(1)}$, $y^{(1)}$, ..., $y^{(t)}$, ... Любое прошлое наблюдение $y^{(i)}$ может повлиять на условное распределение некоторых $y^{(t)}$ (для $t > i$) при условии предыдущих значений. Параметризация графической модели в точном соответствии с эти графом (как в формуле 10.6) может оказаться крайне неэффективной, поскольку число входов и параметров для каждого следующего элемента последовательности будет постоянно возрастать. РНС, обеспечивающая такую же связность, но при этом эффективную параметризацию, показана на рис. 10.8

Интереснее взглянуть на структуру графической модели РНС, получающуюся, если рассматривать скрытые блоки $\boldsymbol{h}^{(t)}$ как случайные величины[1]. Включение скрытых блоков в графическую модель показывает, что РНС дает эффективную параметризацию совместного распределения наблюдений. Предположим, что мы представили совместное распределение дискретных значений в виде таблицы – массива, содержащего по одному элементу для каждой возможной комбинации значений, равному вероятности этой комбинации. Если y может принимать k значений, то в табличном представлении будет $O(k^{\tau})$ параметров. Для сравнения – благодаря разделению число параметров РНС как функция длины последовательности имеет порядок $O(1)$. Число параметров РНС можно изменять для управления емкостью модели, но оно не обязано увеличиваться вместе с длиной последовательности. Из формулы (10.5) видно, что РНС эффективно параметризует долгосрочные связи между переменными, рекуррентно применяя одну и ту же функцию f и одни и те же параметры $\boldsymbol{\theta}$ на каждом временном шаге. На рис. 10.8 показана графическая интерпретация модели. Вершины $\boldsymbol{h}^{(t)}$, включенные в графическую модель, разрывают связь между прошлым и будущим, играя роль посредника между ними. Переменная $y^{(i)}$ в отдаленном прошлом может оказывать влияние на переменную $y^{(t)}$ посредством влияния на $\boldsymbol{h}$. Структура этого графа показывает, что модель можно эффективно параметризовать, используя на каждом шаге одни и те же условные распределения вероятности, и что когда все переменные получены в результате наблюдения, вероятность их совместной комбинации можно эффективно вычислить.

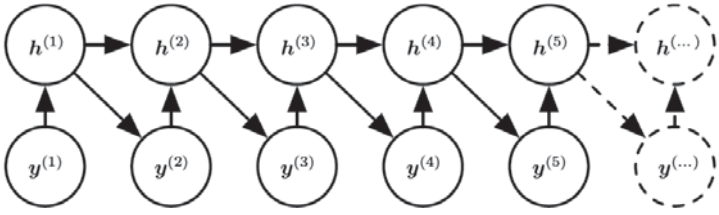

Рис. 10.8 ❖ Введение переменной состояния в графическую модель РНС, хотя она и является детерминированной функцией своих аргументов, помогает понять, как можно получить очень эффективную параметризацию, основанную на формуле (10.5). Все участки этой последовательности (содержащие $\boldsymbol{h}^{(t)}$ и $\boldsymbol{y}^{(t)}$) имеют одинаковую структуру (одно и то же число входов для каждой вершины) и могут разделять параметры с другими участками

Даже в случае эффективной параметризации графической модели некоторые операции остаются вычислительно сложными. Например, трудно предсказать отсутствующие значения в середине последовательности.

За сокращение числа параметров рекуррентным сетям приходится расплачиваться трудностями *оптимизации* этих параметров.

Разделение параметров в рекуррентных сетях основано на предположении о том, что одинаковые параметры можно использовать на разных временных шагах. Это в точности означает, что условное распределение вероятности переменных в момент $t + 1$

[1] Условное распределение таких величин при условии их родителей детерминировано. Это вполне допустимо, хотя графические модели с такими детерминированными скрытыми блоками проектируют нечасто.

при условии переменных в момент t **стационарно**, т. е. соотношение между состоянием системы в предыдущий и в последующий моменты не зависит от t. В принципе, можно было бы считать t дополнительным входом на каждом временном шаге и позволить обучаемой модели выявить временные зависимости между различными шагами. Это было бы гораздо лучше, чем использовать разные условные распределения для каждого t, но тогда сеть должна была бы выполнять экстраполяцию на новые значения t.

Чтобы завершить рассмотрение РНС как графической модели, мы должны еще описать, как производить выборку из модели. Основная интересующая нас операция – выборка примера из условного распределения на каждом временном шаге. Однако имеется одно дополнительное осложнение. У РНС должен быть какой-то механизм определения длины последовательности. Достичь этого можно разными способами.

Если выходом является символ, выбираемый из словаря, то можно включить специальный символ, обозначающий конец последовательности (Schmidhuber, 2012). Если сгенерирован такой символ, то процесс выборки останавливается. В каждом обучающем примере мы вставляем такой символ в качестве дополнительного члена последовательности, сразу после $\boldsymbol{x}^{(\tau)}$.

Другой вариант – ввести в модель дополнительный выход с распределением Бернулли, который говорит, продолжать генерацию после данного временного шага или остановиться. Это более общий подход, чем включение специального символа в словарь, поскольку он применим не только к РНС, порождающей последовательность символов. Например, он годится для РНС, которая выводит последовательность вещественных чисел. Новый выходной блок обычно берут сигмоидным и при его обучении используют перекрестную энтропию в качестве функции потерь. Иначе говоря, сигмоида обучается максимизировать логарифмическую вероятность правильного предсказания окончания последовательности на каждом временном шаге.

Еще один способ определить длину последовательности τ – добавить в модель выход, который предсказывает само целое число τ. Модель сначала выбирает значение τ, а затем данные для τ шагов. При таком подходе необходимо включать дополнительный вход в рекуррентное обновление на каждом временном шаге, чтобы модель знала, подходит она к концу сгенерированной последовательности или еще нет. Этот вход может содержать либо само значение τ, либо число оставшихся шагов $\tau - t$. Без него РНС могла бы генерировать внезапно обрывающиеся последовательности, например неполные предложения. Такой подход основан на разложении

$$P(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}) = P(\tau)P(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)} \mid \tau). \tag{10.34}$$

Стратегия прямого предсказания τ применялась, например, в работе Goodfellow et al. (2014d).

10.2.4. Моделирование контекстно-обусловленных последовательностей с помощью РНС

В предыдущем разделе мы описали соответствие между РНС и ориентированной графической моделью последовательности случайных величин $y^{(t)}$ вообще без входов $\boldsymbol{x}$. Конечно, наша формулировка РНС в виде уравнения (10.8) включает последовательность входов $\boldsymbol{x}^{(1)}$, $\boldsymbol{x}^{(2)}$, ..., $\boldsymbol{x}^{(\tau)}$. В общем случае представление РНС в виде графической модели применимо не только к совместному распределению величин y, но и к условному распределению y при условии $\boldsymbol{x}$. В контексте сетей прямого распространения в разделе 6.2.1.1 уже обсуждалось, что любую модель, представляющую величину $P(\boldsymbol{y}; \boldsymbol{\theta})$, можно интерпретировать как модель, представляющую распределе-

ние $P(\boldsymbol{y} \mid \boldsymbol{\omega})$, где $\boldsymbol{\omega} = \boldsymbol{\theta}$. Мы можем обобщить такую модель на представление распределения $P(\boldsymbol{y} \mid \boldsymbol{x})$, воспользовавшись тем же $P(\boldsymbol{y} \mid \boldsymbol{\omega})$, что и раньше, но сделав $\boldsymbol{\omega}$ функцией от $\boldsymbol{x}$. В случае РНС это можно сделать несколькими способами. Мы рассмотрим самые распространенные и очевидные.

Выше мы обсуждали РНС, которые принимают на входе последовательность векторов $\boldsymbol{x}^{(t)}$ для $t = 1, ..., \tau$. Но можно взять в качестве входа всего один вектор $\boldsymbol{x}$. Если размер вектора фиксирован, то мы можем просто сделать его дополнительным входом РНС, генерирующей последовательность $\mathbf{y}$. Вот несколько типичных способов подать дополнительный вход РНС:

1) в качестве дополнительного входа на каждом временном шаге;
2) в качестве начального состояния $\boldsymbol{h}^{(0)}$;
3) то и другое.

Первый, самый распространенный подход показан на рис. 10.9. Взаимодействие между входом $\boldsymbol{x}$ и каждым векторным скрытым блоком $\boldsymbol{h}^{(t)}$ параметризуется матрицей весов $\boldsymbol{R}$, которой не было в модели, содержащей только последовательность значений y. Одно и то же произведение $\boldsymbol{x}^{\top}\boldsymbol{R}$ добавляется в качестве дополнительного входа скрытых блоков на каждом временном шаге. Выбор $\boldsymbol{x}$ можно рассматривать как определение значения $\boldsymbol{x}^{\top}\boldsymbol{R}$, которое, по существу, является новым параметром смещения, используемым в каждом скрытом блоке. Веса остаются независимыми от входа. Мы можем считать, что эта модель берет параметры $\boldsymbol{\theta}$ безусловной модели и преобразует их в $\boldsymbol{\omega}$, где параметры смещения внутри $\boldsymbol{\omega}$ теперь являются функцией входа.

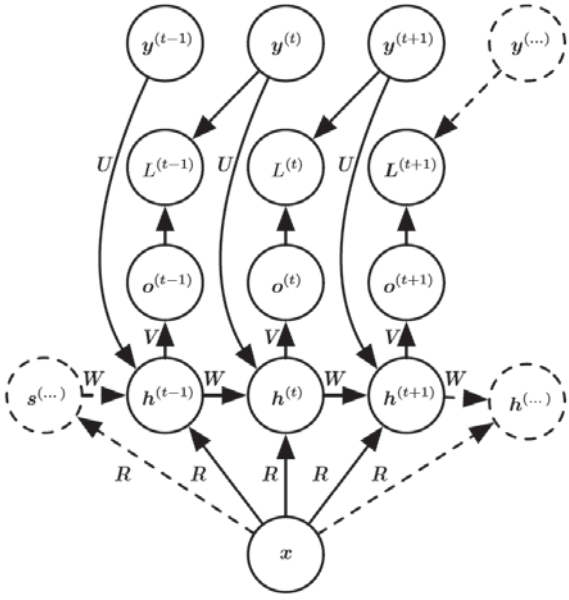

Рис. 10.9 ❖ РНС, отображающая вектор фиксированной длины $\boldsymbol{x}$ в распределение последовательностей $\mathbf{Y}$. Эта РНС подходит для таких задач, как подписывание изображений, когда одно изображение подается на вход модели, порождающей его описание в виде последовательности слов. Каждый элемент $\boldsymbol{y}^{(t)}$ наблюдаемой выходной последовательности служит одновременно входом (для текущего временного шага) и – на этапе обучения – меткой (для предыдущего временного шага)

РНС может получать на входе не единственный вектор $\boldsymbol{x}$, а последовательность векторов $\boldsymbol{x}^{(t)}$. РНС, описываемая уравнением (10.8), соответствует условному распределению $P(\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(\tau)} \mid \boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)})$, для которого справедливо предположение об условной независимости, согласно которому его можно представить в виде произведения

$$\prod_t P(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}). \tag{10.35}$$

Чтобы избавиться от предположения об условной независимости, мы можем добавить связи между выходом в момент t и скрытым блоком в момент $t + 1$, как показано на рис. 10.10. Тогда модель сможет представить произвольное распределение вероятности последовательности $\boldsymbol{y}$. У таких моделей, описывающих распределение вероятности одной последовательности при условии другой последовательности, все же остается одно ограничение: длины обеих последовательностей должны быть равны. Как снять это ограничение, мы покажем в разделе 10.4.

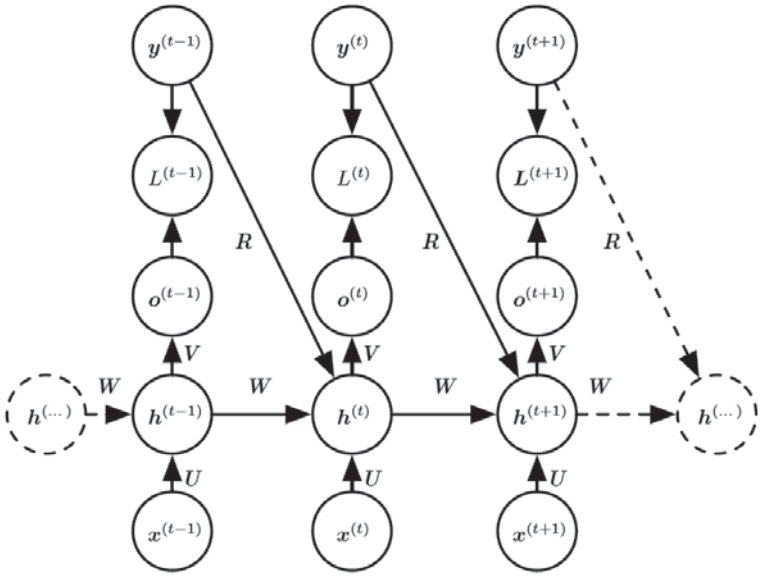

Рис. 10.10 ❖ Условная рекуррентная нейронная сеть, отображающая последовательность значений $\boldsymbol{x}$ переменной длины в распределение последовательностей значений $\boldsymbol{y}$ такой же длины. По сравнению с рис. 10.3, эта РНС содержит связи между предыдущим выходом и текущим состоянием. Эти связи позволяют РНС моделировать произвольное распределение последовательностей $\boldsymbol{y}$ при условии последовательностей $\boldsymbol{x}$ такой же длины. РНС на рис. 10.3 способна представить только распределения, в которых значения $\boldsymbol{y}$ условно независимы друг от друга при условии значений $\boldsymbol{x}$

10.3. Двунаправленные РНС

Все рассмотренные до сих пор рекуррентные сети имели «каузальную» структуру, т. е. на состояние в момент t влияет только информация в прошлые моменты времени

$\boldsymbol{x}^{(1)}$, ..., $\boldsymbol{x}^{(t-1)}$ и текущий вход $\boldsymbol{x}^{(t)}$. Некоторые обсуждавшиеся модели допускают также влияние прошлых значений $\boldsymbol{y}$ на текущее состояние, если значения $\boldsymbol{y}$ доступны.

Но во многих приложениях мы хотим получать предсказание $\boldsymbol{y}^{(t)}$, которое может зависеть от всей входной последовательности. Например, в задаче распознавания речи правильная интерпретация текущего звука как фонемы может зависеть от нескольких следующих фонем из-за коартикуляции и даже от нескольких следующих слов из-за лингвистических зависимостей между соседними словами: если акустически допустимы две интерпретации текущего слова, то, чтобы различить их, возможно, понадобится заглянуть далеко в будущее (или в прошлое). Это относится также к распознаванию рукописных текстов и многих других задач обучения одной последовательности на основе другой, рассматриваемых в следующем разделе.

Двунаправленные рекуррентные нейронные сети были придуманы для удовлетворения именно этой потребности (Schuster and Paliwal, 1997). Они оказались чрезвычайно успешными (Graves, 2012) в распознавании рукописных текстов (Graves et al., 2008; Graves and Schmidhuber, 2009), распознавании речи (Graves and Schmidhuber, 2005; Graves et al., 2013) и биоинформатике (Baldi et al., 1999).

Как следует из самого названия, двунаправленные РНС являются комбинацией РНС, движущейся вперед во времени (от начала последовательности к ее концу), и РНС, движущейся в обратном направлении. На рис. 10.11 показана типичная двунаправленная РНС; $\boldsymbol{h}^{(t)}$ обозначает состояние той РНС, что движется вперед, а $\boldsymbol{g}^{(t)}$ – той, что движется назад. Это позволяет выходным блокам $\boldsymbol{o}^{(t)}$ вычислять представление, которое зависит *как от прошлого, так и от будущего*, но наиболее чувствительно к входным значениям вблизи момента t; при этом задавать окно фиксированного размера вокруг t необязательно (в отличие от сетей прямого распространения, сверточных сетей и обычных РНС с буфером предвыборки фиксированного размера).

Эта идея естественно обобщается на двумерные входные данные, например изображения, для чего нужны *четыре* РНС, по одной в каждом направлении: вверх, вниз, влево, вправо. Тогда в каждой точке (i,j) двумерной сети выходной блок $O_{i,j}$ мог бы вычислять представление, которое улавливает в основном локальную информацию, но может зависеть и от удаленных входов, если РНС способна такую информацию нести. По сравнению со сверточной сетью, применение РНС к изображениям обходится дороже, зато может учитывать дальние боковые взаимодействия между элементами одной карты признаков (Visin et al., 2015; Kalchbrenner et al., 2015). Действительно, уравнения прямого распространения для таких РНС можно записать в виде, показывающем, что в них используется свертка, которая вычисляет вход снизу в каждый слой, а только потом производится рекуррентное распространение поперек карты признаков, учитывающее боковые взаимодействия.

10.4. Архитектуры кодировщик-декодер или последовательность в последовательность

На рис. 10.5 мы видели, что РНС может отобразить входную последовательность в вектор фиксированного размера. На рис. 10.9 было показано, что РНС может отобра1зить вектор фиксированного размера в последовательность, а на рис. 10.3, 10.4, 10.10 и 10.11 – что РНС способна отобразить входную последовательность на выходную такой же длины.

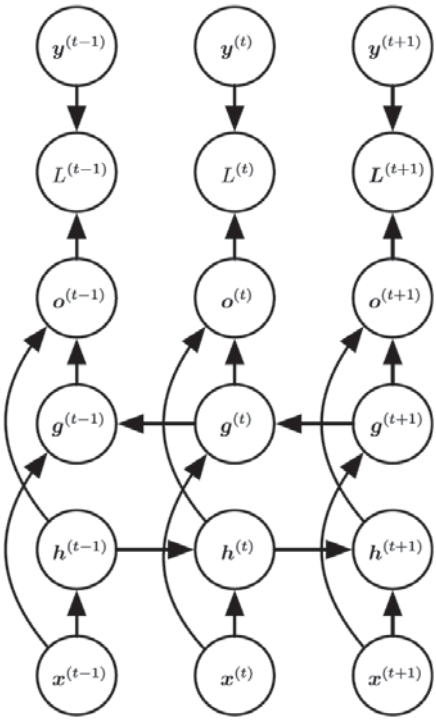

Рис. 10.11 ❖ Вычисление типичной двунаправленной рекуррентной нейронной сети, которая должна обучиться отображать входные последовательности $\boldsymbol{x}$ на выходные последовательности $\boldsymbol{y}$, с функцией потерь $L^{(t)}$ на каждом шаге t. Рекуррентные блоки $\boldsymbol{h}$ распространяют информацию вперед во времени (слева направо), а блоки $\boldsymbol{g}$ – назад во времени (справа налево). Таким образом, в каждой точке t выходные блоки $\boldsymbol{o}^{(t)}$ могут пользоваться сводной информацией о прошлом из входа $\boldsymbol{h}^{(t)}$ и сводной информацией о будущем из входа $\boldsymbol{g}^{(t)}$

В этом разделе мы обсудим, как обучить РНС отображать входную последовательность на выходную необязательно такой же длины. Такая задача возникает во многих приложениях, например распознавании речи, машинном переводе и вопросно-ответных системах, где входные и выходные последовательности в обучающем наборе, вообще говоря, имеют разную длину (хотя их длины могут быть взаимосвязаны).

Вход такой РНС часто называют «контекстом». Мы хотим породить представление контекста C. Контекст C может быть вектором или последовательностью векторов, агрегирующей входную последовательность $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(n_x)})$.

Простейшая архитектура РНС для отображения одной последовательности переменной длины на другую была предложена сначала в работе Cho et al. (2014a), а вскоре вслед за ней в работе Sutskever et al. (2014), авторы которой независимо разработали эту архитектуру и впервые применили ее к машинному переводу. Первая система основана на оценке предложений, генерируемых другой системой машинного перевода, а во второй используется автономная рекуррентная сеть для генерации переводов. Авторы назвали эту архитектуру (рис. 10.12) кодировщик-декодер (encoder-decoder) или последовательность в последовательность (sequence-to-sequence). Идея очень проста:

(1) **Кодировщик** или **читатель**, или **входная** РНС обрабатывает входную последовательность. Кодировщик порождает контекст C, обычно в виде простой функции конечного скрытого состояния. (2) **Декодер**, или **писатель**, или **выходная** РНС обусловлена этим вектором фиксированной длины (как на рис. 10.9) и генерирует выходную последовательность $\boldsymbol{Y} = (\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(n_y)})$. От представленных выше в этой главе архитектур эта отличается тем, что длины n_x и n_y могут различаться, тогда как в предыдущих архитектурах действовало ограничение $n_x = n_y = \tau$. В архитектуре последовательность в последовательность обе РНС совместно обучаются максимизировать среднее величины $\log P(\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(n_y)}) \mid \boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(n_x)})$ по всем парам последовательностей $\boldsymbol{x}$ и $\boldsymbol{y}$ в обучающем наборе. Последнее состояние $\boldsymbol{h}_{n_x}$ кодирующей РНС обычно используется как представление C входной последовательности, подаваемой на вход декодирующей РНС.

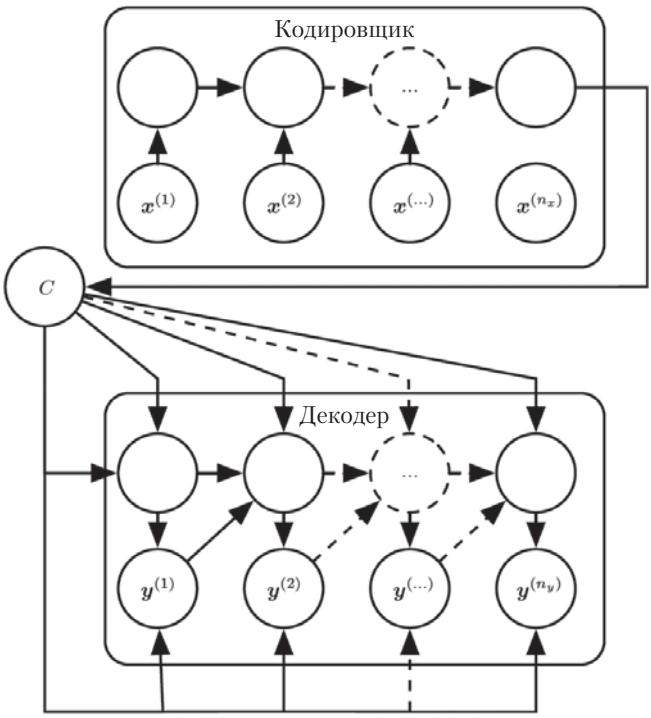

Рис. 10.12 ❖ Пример архитектуры РНС типа кодировщик-декодер, или последовательность в последовательность, которая обучается генерировать выходную последовательность $\mathbf{y}^{(1)}, ..., \mathbf{y}^{(n_y)}$ по входной последовательности $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_x)}$. Сеть состоит из кодирующей РНС, которая читает входную последовательность, и декодирующей РНС, которая генерирует выходную последовательность (или вычисляет вероятность заданной выходной последовательности). Конечное скрытое состояние кодирующей РНС используется для вычисления контекстной переменной C фиксированного размера, которая представляет собой семантическую сводку входной последовательности и подается на вход декодирующей РНС

Если контекстом C является вектор, то декодирующая РНС – обычная РНС, отображающая вектор в последовательность, описанная в разделе 10.2.4. Как мы ви-

дели, существуют, по меньшей мере, два способа организовать вход такой РНС. Входом можно считать начальное состояние РНС, или вход можно соединять со скрытыми блоками на каждом временном шаге. Можно также сочетать оба способа.

Не требуется, чтобы размеры скрытого слоя кодировщика и декодера совпадали.

Очевидное ограничение этой архитектуры проявляется, когда размер контекста C, порождаемого кодировщиком, слишком мал для формирования надлежащей сводки длинной последовательности. Это явление отмечено в работе Bahdanau et al. (2015) в применении к машинному переводу. Авторы предложили сделать C последовательностью переменной длины, а не вектором фиксированной длины. Кроме того, они ввели **механизм внимания**, который обучается ассоциировать элементы последовательности C с элементами выходной последовательности. Детали см. в разделе 12.4.5.1.

10.5. Глубокие рекуррентные сети

Вычисления в большинстве РНС можно разложить на три блока параметров и ассоциированные с ними преобразования:

1) из входа в скрытое состояние;
2) из предыдущего скрытого состояния в следующее;
3) из скрытого состояния в выход.

В архитектуре РНС, показанной на рис. 10.3, с каждым из этих трех блоков ассоциирована одна матрица весов. Иными словами, при развертке сети каждый блок будет соответствовать мелкому преобразованию. Под «мелким» мы понимаем преобразование, которое было бы представлено одним слоем в глубоком МСП. Как правило, это обученное аффинное преобразование, за которым следует фиксированная нелинейность.

Даст ли какой-нибудь выигрыш наделение этих операций глубиной? Эксперименты (Graves et al., 2013; Pascanu et al., 2014a) уверенно свидетельствуют в пользу такого предположения. Экспериментальные факты согласуются с идеей о том, что для выполнения требуемых отображений нужна достаточная глубина. См. также более ранние работы по глубоким РНС Schmidhuber (1992), El Hihi and Bengio (1996) и Jaeger (2007a).

В работе Graves et al. (2013) впервые продемонстрировано значительное преимущество от разложения состояния РНС в несколько слоев, как на рис. 10.13a. Можно считать, что нижние слои в иерархии, показанной на рис. 10.13a, играют роль в преобразовании входных данных в представление, более подходящее для верхних уровней скрытого состояния. В работе Pascanu et al. (2014a) сделан следующий шаг: предложено включать отдельный МСП (возможно, глубокий) для каждого из трех перечисленных выше блоков, как показано на рис. 10.13b. По соображениям репрезентативной емкости, кажется естественным наделить каждый из трех шагов достаточно большой емкостью, но если для этого увеличивать глубину, то обучение может осложниться из-за трудностей оптимизации. В общем случае оптимизировать проще более мелкие архитектуры, а увеличение глубины на рис. 10.13b приводит к удлинению кратчайшего пути от переменной на шаге t к переменной на шаге $t + 1$. Например, если для перехода состояний используется МСП с одним скрытым слоем, то длина кратчайшего пути между переменными на любых двух временных шагах удваивается, по сравнению с обычной РНС на рис. 10.3. Однако в работе Pascanu et al. (2014a) отмечено, что эту проблему можно сгладить путем добавления прямых связей внутри скрытого слоя, как показано на рис. 10.13c.

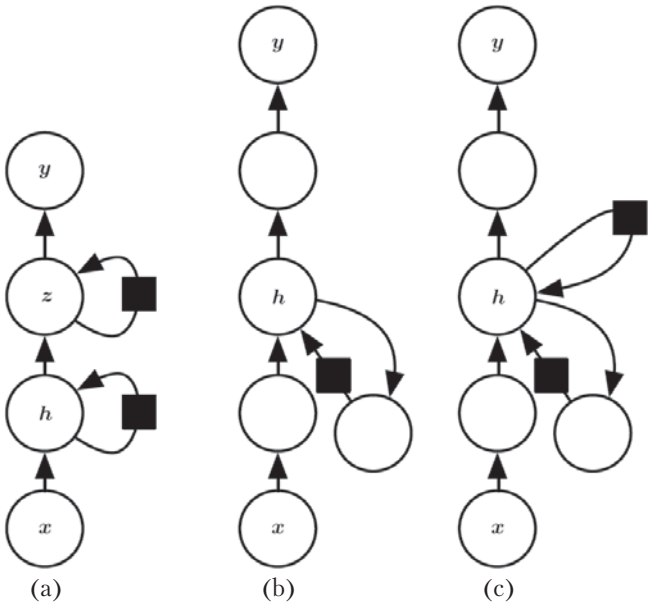

Рис. 10.13 ❖ Рекуррентную нейронную сеть можно сделать глубокой разными способами (Pascanu et al., 2014a). (*a*) Скрытое рекуррентное состояние можно разделить на иерархически организованные группы. (*b*) Между входом и скрытым состоянием, между двумя скрытыми уровнями скрытого состояния и между скрытым состоянием и выходом можно поместить более глубокое вычисление (например, МСП). Это может удлинить кратчайший путь, соединяющий разные временные шаги. (*c*) Эффект удлинения пути можно сгладить путем добавления прямых связей

10.6. Рекурсивные нейронные сети

Рекурсивные нейронные сети[1] – еще один вид обобщения рекуррентных сетей, для которого характерен граф вычислений, структурированный как глубокое дерево, а не как цепная структура, присущая РНС. Типичный граф вычислений для рекурсивной сети показан на рис. 10.14. Рекурсивные нейронные сети были введены в работе Pollack (1990), а их потенциальное применение к обучению рассуждениям описано в работе Bottou (2011). Рекурсивные сети успешно применялись для обработки структур данных, используемых в качестве входа нейронной сети (Frasconi et al., 1997, 1998), в обработке естественных языков (Socher et al., 2011a,c, 2013a) и в компьютерном зрении (Socher et al., 2011b).

Одно очевидное преимущество рекурсивных сетей по сравнению с рекуррентными – тот факт, что для последовательности одной и той же длины τ глубину (измеренную как количество композиций нелинейных операций) можно резко снизить с τ до $O(\log \tau)$, это может оказаться весьма полезно при работе с долгосрочными зависимостями. Открытый вопрос – как лучше всего структурировать дерево. Один из вариантов – выбрать структуру, не зависящую от данных, например сбалансированное

[1] Мы не употребляем аббревиатуру РНС для рекурсивных нейронных сетей во избежание путаницы с рекуррентными нейронными сетями.

двоичное дерево. В некоторых предметных областях подходящая структура диктуется внешними соображениями. Например, при обработке предложений естественного языка в качестве структуры дерева рекурсивной сети можно взять структуру дерева грамматического разбора предложения, которое строит анализатор языка (Socher et al., 2011a, 2013a). В идеале хотелось бы, чтобы обучаемая модель сама выводила структуру дерева, соответствующую любому заданному входу, как предложено в работе Bottou (2011).

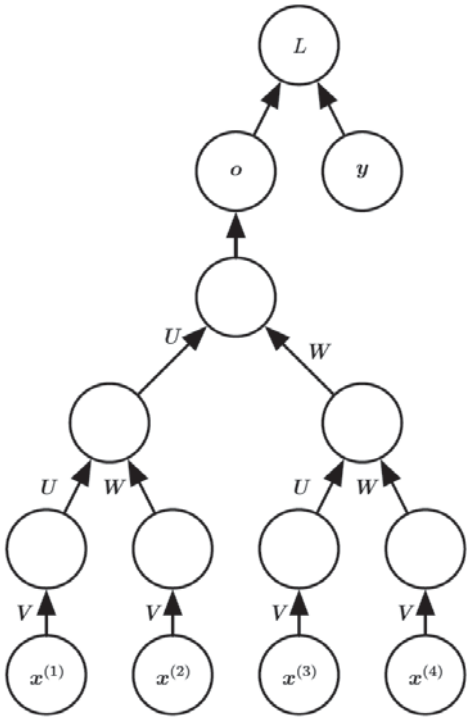

Рис. 10.14 ❖ Граф вычислений рекурсивной сети является деревом, а не цепочкой, как в рекуррентных сетях. Последовательность переменной длины $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ можно отобразить на представление фиксированного размера (выход $\boldsymbol{o}$) с фиксированным набором параметров (матрицы весов $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). На рисунке показан случай обучения с учителем, когда предъявляется метка $\boldsymbol{y}$, ассоциированная со всей последовательностью

У идеи рекурсивной сети много вариантов. Например, в работах Frasconi et al. (1997) и Frasconi et al. (1998) со структурой дерева ассоциируются данные, а с отдельными узлами дерева – входы и метки. Необязательно, чтобы в каждом узле выполнялись традиционные для искусственного нейрона вычисления (аффинное преобразование всех выходов, за которым следует монотонная нелинейность). Например, в работе Socher et al. (2013a) предложено использовать тензорные операции и билинейные формы, которые раньше с успехом применялись для моделирования связей между концептами (Weston et al., 2010; Bordes et al., 2012), представленными непрерывными векторами (вложениями).

10.7. Проблема долгосрочных зависимостей

Математическая проблема обучения долгосрочных зависимостей в рекуррентных сетях описана в разделе 8.2.5. Основная трудность состоит в том, что градиенты, распространяющиеся через много слоев, либо исчезают (в большинстве случаев), либо начинают взрывообразно расти (редко, но с большим уроном для оптимизации). Даже если предположить, что при заданных параметрах рекуррентная сеть устойчива (может хранить воспоминания без взрывного роста градиентов), все равно остается проблема назначения долгосрочным зависимостям экспоненциально меньших (из-за перемножения большого числа якобианов) весов, чем краткосрочным. Во многих источниках этот вопрос освещается более глубоко (Hochreiter, 1991; Doya, 1993; Bengio et al., 1994; Pascanu et al., 2013). В этом разделе мы подробнее опишем саму проблему, а в последующих – подходы к ее преодолению.

В рекуррентных сетях композиция одной и той же функции вычисляется многократно – по одному разу на каждом временном шаге. Это может приводить к поведению, весьма далекому от линейного (рис. 10.15).

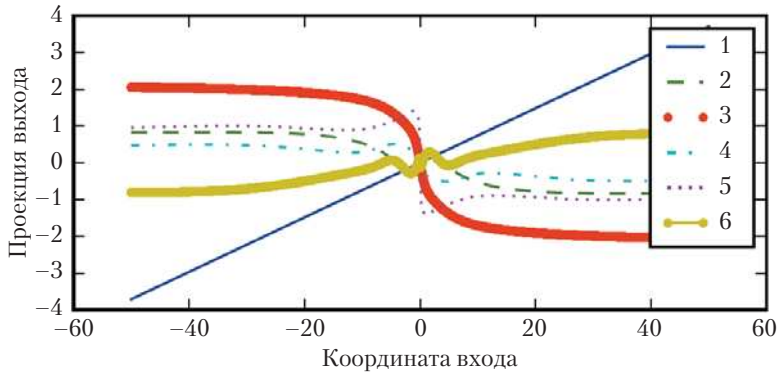

Рис. 10.15 ❖ Повторная композиция функций. Многократная композиция нелинейной функции (например, показанного здесь гиперболического тангенса) приводит к сильно нелинейному результату; обычно в большинстве точек производная очень мала, в некоторых велика, и часто наблюдается переход от возрастания к убыванию и наоборот. На этом рисунке показана линейная проекция 100-мерного скрытого состояния на одно измерение, отложенное по оси *y*. По оси *x* отложена координата начального состояния вдоль случайно выбранного направления в 100-мерном пространстве. Таким образом, этот график можно рассматривать как сечение графика многомерной функции. На графиках показана функция после каждого временного шага, или, эквивалентно, результат многократной композиции функции перехода с самой собой

В частности, композиция функция, применяемая в рекуррентных нейронных сетях, чем-то напоминает умножение матриц. Рекуррентное соотношение

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^{\top}\boldsymbol{h}^{(t-1)} \tag{10.36}$$

можно рассматривать как очень простую РНС без функции активации и без входов $\boldsymbol{x}$. В разделе 8.2.5 было отмечено, что это соотношение, по существу, описывает возведение в степень. Его можно упростить следующим образом:

$$\boldsymbol{h}^{(t)} = (\boldsymbol{W}^t)^\top \boldsymbol{h}^{(0)}, \tag{10.37}$$

а если $\boldsymbol{W}$ допускает спектральное разложение вида

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top, \tag{10.38}$$

где $\boldsymbol{Q}$ – ортогональная матрица, то это соотношение можно еще упростить:

$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q}\boldsymbol{h}^{(0)}. \tag{10.39}$$

Собственные значения возводятся в степень t, в результате чего собственные значения, которые по абсолютной величине меньше 1, стремятся к нулю, а те, что больше 1, резко возрастают. В конечном итоге компоненты $\boldsymbol{h}^{(0)}$, не сонаправленные наибольшему собственному вектору, будут отброшены.

Эта проблема особенно остро стоит для рекуррентных сетей. Возьмем скалярный случай и представим себе многократное умножение веса w на себя. В зависимости от абсолютной величины w произведение w^t будет стремиться либо к нулю, либо к бесконечности. Если построить нерекуррентную сеть с различными весами $w^{(t)}$ на каждом временном шаге, то ситуация будет иной. Если начальное состояние равно 1, то состояние в момент t равно произведению $\prod_t w^{(t)}$. Предположим, что значения $w^{(t)}$ генерируются случайным образом независимо друг от друга с нулевым средним и дисперсией v. Тогда дисперсия произведения равна $O(v^n)$. Чтобы получить желаемую дисперсию v^*, мы можем подобрать индивидуальные веса, так чтобы их дисперсия была равна $v = \sqrt[n]{v^*}$. Таким образом, даже в очень глубоких сетях за счет тщательно подобранного масштабирования можно избежать проблемы исчезающего и взрывного градиента (см. Sussillo (2014)).

Проблема исчезающего и взрывного градиента для РНС была независимо обнаружена несколькими исследователями (Hochreiter, 1991; Bengio et al., 1993, 1994). Можно было бы надеяться избежать ее, просто оставаясь в области пространства параметров, где градиенты не исчезают и не растут взрывообразно. К сожалению, для хранения «воспоминаний» способом, устойчивым к малым возмущениям, РНС должна войти в область пространства параметров, где градиенты исчезают (Bengio et al., 1993, 1994). Точнее говоря, если модель способна представить долгосрочные зависимости, то абсолютная величина градиента долгосрочного взаимодействия экспоненциально меньше, чем краткосрочного. Это не означает, что сеть вообще невозможно обучить, просто обучение долгосрочных зависимостей может занять очень много времени, потому что сигнал об этих зависимостях будет замаскирован мельчайшими флуктуациями, возникающими из-за краткосрочных зависимостей. Эксперименты, описанные в работе Bengio et al. (1994), показывают, что на практике по мере увеличения протяженности зависимостей, которые требуется запоминать, градиентная оптимизация становится все труднее, и вероятность успешно обучить традиционную РНС методом стохастического градиентного спуска быстро спадает до 0, когда длина последовательностей равна всего 10 или 20.

Более глубокое рассмотрение рекуррентных сетей как динамических систем см. в работах Doya (1993), Bengio et al. (1994), Siegelmann and Sontag (1995), а обзор литературы – в работе Pascanu et al. (2013). Далее в этой главе мы рассмотрим различные подходы, предложенные с целью уменьшить трудность обучения долгосрочных зависимостей (иногда удается обучить РНС зависимостям, существующим на протяжении сотен шагов), однако отметим, что эта проблема остается одной из главных в машинном обучении.

10.8. Нейронные эхо-сети

Отображение рекуррентных весов $\boldsymbol{h}^{(t-1)}$ в $\boldsymbol{h}^{(t)}$ и отображение входов на веса – $\boldsymbol{x}^{(t)}$ в $\boldsymbol{h}^{(t)}$ – параметры рекуррентной сети, которые труднее всего поддаются обучению. Для борьбы с этими трудностями было предложено (Jaeger, 2003; Maass et al., 2002; Jaeger and Haas, 2004; Jaeger, 2007b) задавать рекуррентные веса так, чтобы скрытые рекуррентные блоки запоминали историю прошлых входов, а *обучать только выходные веса*. Эта идея была независимо предложена для **эхо-сетей** (echo state networks), или ESN (Jaeger and Haas, 2004; Jaeger, 2007b), и **машин неустойчивых состояний** (liquid state machines) (Maass et al., 2002). Обе модели похожи, но в первой используются скрытые блоки с непрерывными значениями, а во второй – импульсные нейроны (с бинарным выходом). И эхо-сети, и машины неустойчивых состояний объединены общим названием **резервуарные вычисления** (reservoir computing) (Lukoševicius and Jaeger, 2009), отражающим тот факт, что скрытые блоки образуют резервуар временных признаков, которые могут запоминать различные аспекты истории входов.

В некотором смысле такие рекуррентные сети с резервуарными вычислениями напоминают ядерные методы: они отображают последовательность произвольной длины (историю входов до момента t) на вектор фиксированной длины (рекуррентное состояние $\boldsymbol{h}^{(t)}$), к которому можно применить линейный предиктор (обычно линейную регрессию) для решения интересующей проблемы. В таком случае критерий обучения легко спроектировать в виде выпуклой функции выходных весов. Например, если выход получается в результате линейной регрессии скрытых входов на выходные метки, а критерий обучения – среднеквадратическая ошибка, то функция выпуклая, и задачу можно надежно решить с помощью простых алгоритмов обучения (Jaeger, 2003).

Следовательно, возникает важный вопрос: как задать вход и рекуррентные веса, чтобы в состоянии рекуррентной нейронной сети можно было представить достаточно полную историю? В литературе по резервуарным вычислениям предлагается рассматривать рекуррентную сеть как динамическую систему и задавать входы и веса, так чтобы эта система была близка к устойчивости.

Первоначальная идея состояла в том, чтобы сделать собственные значения якобиана функции перехода состояний близкими к 1. В разделе 8.2.5 говорилось, что важной характеристикой рекуррентной сети является спектр собственных значений якобианов $\boldsymbol{J}^{(t)} = (\partial s^{(t)} / \partial s^{(t-1)})$. Особый интерес представляет **спектральный радиус** $\boldsymbol{J}^{(t)}$, определяемый как максимум абсолютных величин собственных значений.

Чтобы понять, на что влияет спектральный радиус, рассмотрим простой случай обратного распространения, в котором матрица Якоби $\boldsymbol{J}$ не зависит от t. Так бывает, например, когда сеть чисто линейная. Предположим, что у $\boldsymbol{J}$ имеется собственный вектор $\boldsymbol{v}$, соответствующий собственному значению λ. Посмотрим, что происходит, когда вектор градиента распространяется назад во времени. Если начать с градиента $\boldsymbol{g}$, то после одного шага обратного распространения градиент будет равен $\boldsymbol{Jg}$, а после n шагов – $\boldsymbol{J}^n\boldsymbol{g}$. Теперь посмотрим, что случится, если обратному распространению подвергнуть возмущенный вектор $\boldsymbol{g}$. Если начать с $\boldsymbol{g} + \delta\boldsymbol{v}$, то после одного шага получится $\boldsymbol{J}(\boldsymbol{g} + \delta\boldsymbol{v})$, а после n шагов – $\boldsymbol{J}^n(\boldsymbol{g} + \delta\boldsymbol{v})$. Отсюда видно, что результаты обратного распространения, начавшегося с $\boldsymbol{g}$ и с $\boldsymbol{g} + \delta\boldsymbol{v}$, после n шагов расходятся на $\delta\boldsymbol{J}^n\boldsymbol{v}$. Если в качестве $\boldsymbol{v}$ взять единичный собственный вектор $\boldsymbol{J}$ с собственным значением λ, то умножение на якобиан просто масштабирует разность на каждом шаге. Результаты

двух описанных выше выполнений обратного распространения разделены расстоянием $\delta|\lambda|^n$: если $\boldsymbol{v}$ соответствует наибольшему значению $|\lambda|$, то это возмущение дает наибольшее возможное расхождение с начальным возмущением δ.

Если $|\lambda| > 1$, то величина расхождения $\delta|\lambda|^n$ экспоненциально возрастает, если $|\lambda| < 1$, то экспоненциально убывает.

Разумеется, в этом примере предполагалось, что якобиан одинаков на всех временных шагах, что соответствует рекуррентной нейронной сети без нелинейностей. Если же нелинейность присутствует, то ее производная после многих шагов будет близка к нулю, что позволит предотвратить взрывообразный рост из-за большого спектрального радиуса. На самом деле в большинстве недавних работ по эхо-сетям предлагается использовать спектральный радиус, много больший 1 (Yildiz et al., 2012; Jaeger, 2012).

Все сказанное выше об обратном распространении посредством повторного умножения на матрицу равным образом применимо и к прямому распространению в сети без нелинейностей, когда состояние $\boldsymbol{h}^{(t+1)} = \boldsymbol{h}^{(t)\top}\boldsymbol{W}$.

Если линейное отображение $\boldsymbol{W}^\top$ всегда уменьшает $\boldsymbol{h}$ по норме L^2, то говорят, что отображение **сжимающее**. Если спектральный радиус меньше 1, то отображение $\boldsymbol{h}^{(t)}$ в $\boldsymbol{h}^{(t+1)}$ сжимающее, поэтому небольшое изменение с каждым шагом становится все меньше. Поэтому сеть неизбежно забывает информацию о прошлом, если для хранения вектора состояния используются вычисления конечной точности (например, с 32-разрядными целыми).

Матрица Якоби говорит, как малое изменение $\boldsymbol{h}^{(t)}$ распространяется на один шаг вперед, или, эквивалентно, как градиент $\boldsymbol{h}^{(t+1)}$ распространяется на один шаг назад в ходе обратного распространения. Отметим, что ни $\boldsymbol{W}$, ни $\boldsymbol{J}$ не обязаны быть симметричными (хотя обе матрицы квадратные и вещественные), поэтому они могут иметь комплексные собственные значения, мнимые части которых соответствуют потенциально колебательному поведению (при повторном применении одного и того же якобиана). Хотя $\boldsymbol{h}^{(t)}$ или небольшая вариация $\boldsymbol{h}^{(t)}$ при обратном распространении принимают вещественные значения, их можно выразить в таком комплексном базисе. Важно лишь, что происходит с абсолютной величиной (модулем комплексного числа) координат в потенциально комплексном базисе при умножении матрицы на вектор. Собственные значения, абсолютная величина которых больше 1, соответствуют увеличению (при повторном применении – экспоненциальному росту) или уменьшению (экспоненциальному убыванию).

В случае нелинейного отображения якобиан может изменяться на каждом шаге. Поэтому динамика становится более сложной. Но по-прежнему небольшое изменение может стать большим после нескольких шагов. Одно из различий между линейным и нелинейным случаями состоит в том, что «сплющивающая» нелинейность типа tanh может сделать рекуррентную динамику ограниченной. Отметим, что динамика обратного распространения может оставаться неограниченной, даже если прямое распространение имеет ограниченную динамику, например когда все tanh-блоки, принадлежащие последовательности, находятся в середине своего почти линейного участка и связаны матрицами весов со спектральными радиусами больше 1. Впрочем, редко случается, что все tanh-блоки одновременно оказываются на линейном участке активации.

В эхо-сетях принята простая стратегия – зафиксировать веса, так чтобы спектральный радиус был равен примерно 3, тогда информация переносится вперед во времени, но взрывного роста нет из-за стабилизирующего влияния насыщающих нелинейностей типа tanh.

Сравнительно недавно было показано, что методы, используемые для задания весов в эхо-сетях, можно применять и для *инициализации* весов в рекуррентных сетях полного обучения (когда рекуррентные веса между скрытыми слоями обучаются с помощью обратного распространения во времени), что помогает обучаться долгосрочным зависимостям (Sutskever, 2012; Sutskever et al., 2013). В этой конфигурации хорошие результаты дает выбор начального спектрального радиуса 1.2 в сочетании со схемой разреженной инициализации, описанной в разделе 8.4.

10.9. Блоки с утечками и другие стратегии нескольких временных масштабов

Один из способов включения долгосрочных зависимостей – спроектировать модель, работающую в нескольких временных масштабах, так что одни части модели работают в мелком масштабе и могут обрабатывать мелкие детали, а другие – работающие в крупном масштабе – эффективно передают информацию из отдаленного прошлого в настоящее. Существуют различные стратегии построения мелких и крупных масштабов: добавление прямых связей сквозь время; «блоки с утечками», которые интегрируют сигналы с разными временными постоянными; удаление некоторых связей, используемых для моделирования мелких масштабов.

10.9.1. Добавление прямых связей сквозь время

Для получения грубого временного масштаба можно добавить прямые связи между переменными в отдаленном прошлом и переменными в настоящем. Идея таких прямых связей восходит к работе Lin et al. (1996) и вытекает из идеи включения задержек в нейронные сети прямого распространения (Lang and Hinton, 1988). В обыкновенной рекуррентной сети рекуррентная связь идет из блока в момент t к блоку в момент $t + 1$. Но можно построить рекуррентные сети с большими задержками (Bengio, 1991).

В разделе 8.2.5 мы видели, что градиенты могут исчезать или экспоненциально расти *при увеличении числа шагов*. Для смягчения этой проблемы в работе Lin et al. (1996) введены рекуррентные соединения с временной задержкой d. Теперь градиент экспоненциально убывает как функция τ/d, а не τ. Поскольку имеются как связи с задержкой, так и одношаговые связи, градиенты все же могут экспоненциально расти относительно τ. Это позволяет алгоритму обучения улавливать долгосрочные зависимости, хотя и не все такие зависимости хорошо представляются подобным образом.

10.9.2. Блоки с утечкой и спектр разных временных масштабов

Еще один способ получить пути, на которых произведение производных близко к 1, – включать блоки с линейными соединениями с самими собой (самосоединениями), имеющими веса, близкие к единице.

В формуле вычисления скользящего среднего $\mu^{(t)}$ некоторой величины $v^{(t)}$: $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$ параметр α является примером линейного самосоединения $\mu^{(t-1)}$ с $\mu^{(t)}$. Если α близко к 1, то скользящее среднее помнит информацию о прошлом на протяжении долгого времени, а если α близко к 0, то информация о прошлом быстро забывается. Скрытые блоки с линейными самосоединениями могут вести себя аналогично скользящему среднему и называются **блоками с утечкой**, или **протекающими**.

Прямые связи с пропуском d временных шагов гарантируют, что блок сможет обучиться влиянию значений, находящихся от него на d шагов в прошлом. Использование линейных самосоединений с весом, близким к 1, – другой способ обеспечить блоку доступ к прошлым значениям. И этот подход допускает более плавную и гибкую настройку путем изменения вещественного параметра α, а не целочисленной длины пропуска.

Эти идеи предложены в работах Mozer (1992) и by El Hihi and Bengio (1996). Блоки с утечкой оказались полезны также в контексте эхо-сетей (Jaeger et al., 2007).

Существуют две основные стратегии задания временных констант для настройки блоков с утечкой. Одна – вручную зафиксировать их значения, например выбрав их из некоторого распределения один раз на этапе инициализации. Вторая – сделать константы свободными параметрами и обучить их. Похоже, что включение блоков с утечкой в разных временных масштабах помогает справиться с долгосрочными зависимостями (Mozer, 1992; Pascanu et al., 2013).

10.9.3. Удаление связей

Еще один подход к обработке долгосрочных зависимостей – организация состояния РНС в нескольких временных масштабах (El Hihi and Bengio, 1996) с целью упростить протекание информации на большое расстояние в более медленном масштабе.

Эта идея отличается от прямых связей сквозь время, поскольку подразумевается активное *удаление* связей длины 1 и замена их более длинными. Модифицированные таким способом блоки вынуждены работать в более протяженном временном масштабе. Прямые связи *добавляют* ребра. Блоки, получившие новые связи, могут обучиться работе в протяженном временном масштабе, но могут поступить и наоборот, предпочтя другие, более короткие связи.

Есть несколько способов принудить группу рекуррентных блоков к работе в разных временных масштабах. Один из них – сделать рекуррентные блоки протекающими, но ассоциировать разные группы блоков с разными фиксированными временными маштабами. Эта идея предложена в работе Mozer (1992) и успешно воплощена в работе Pascanu et al. (2013). Другой способ – производить явные дискретные обновления в разные моменты времени с разной частотой для разных групп блоков. Этот подход применен в работах El Hihi and Bengio (1996) и Koutnik et al. (2014). Он показал хорошие результаты на ряде эталонных наборов данных.

10.10. Долгая краткосрочная память и другие вентильные РНС

На момент написания книги самыми эффективными моделями последовательностей в практических приложениях были **вентильные РНС** (gated RNN). К ним относятся **долгая краткосрочная память** (long short-term memory – LSTM) и сети, основанные на вентильных рекуррентных блоках.

Как и блоки с утечкой, вентильные РНС основаны на идее прокладывания таких путей сквозь время, на которых производные не обнуляются и не устремляются резко вверх. В случае блоков с утечкой это достигалось с помощью весов связей – постоянных, задаваемых вручную или являющихся параметрами. Вентильные РНС обобщают эту идею на веса связей, которые могут изменяться на каждом временном шаге.

Блоки с утечкой позволяют сети *накапливать* информацию (например, свидетельства в пользу конкретного признака или категории) на протяжении длительного времени. Но иногда полезно, чтобы после использования этой информации нейронная сеть забыла старое состояние. Например, если последовательность состоит из подпоследовательностей и мы хотим, чтобы блок с утечкой накапливал свидетельства внутри каждой подпоследовательности, то необходим механизм забывания старого состояния – сброса его в нуль. И хорошо бы, чтобы нейронная сеть обучилась, когда это нужно делать, не обременяя нас принятием решения. Именно для этого и предназначены вентильные РНС.

10.10.1. Долгая краткосрочная память

Удачная мысль о введении петель для создания путей, по которым градиент может течь длительное время, – основной вклад в первоначальную модель долгой краткосрочной памяти (Hochreiter and Schmidhuber, 1997). Позднее было внесено важнейшее дополнение – вес петли должен быть контекстно-обусловленным, а не фиксированным (Gers et al., 2000). Сделав вес петли вентильным (управляемым другим скрытым блоком), мы можем динамически изменять временной масштаб интегрирования. В данном случае имеется в виду, что даже для LSTM с фиксированными параметрами временной масштаб интегрирования может изменяться в зависимости от входной последовательности, поскольку временные константы выводятся самой моделью. Идея LSTM оказалась чрезвычайно успешной во многих приложениях, например: неограниченное распознавание рукописных текстов (Graves et al., 2009), распознавание речи (Graves et al., 2013; Graves and Jaitly, 2014), порождение рукописных текстов (Graves, 2013), машинный перевод (Sutskever et al., 2014), подписывание изображений (Kiros et al., 2014b; Vinyals et al., 2014b; Xu et al., 2015) и грамматический разбор (Vinyals et al., 2014a).

Принципиальная схема LSTM показана на рис. 10.16. Ниже приведены соответствующие уравнения прямого распространения для архитектуры мелкой рекуррентной сети. Есть также примеры успешного использования более глубоких архитектур (Graves et al., 2013; Pascanu et al., 2014a). Вместо блока, который просто применяет поэлементную нелинейность к аффинному преобразованию входов и рекуррентным блокам, в рекуррентных LSTM-сетях имеются «LSTM-ячейки», обладающие внутренней рекуррентностью (петлей) в дополнение к внешней рекуррентности РНС. У каждой ячейки такие же входы и выходы, как у обыкновенной рекуррентной сети, но еще имеются дополнительные параметры и система вентильных блоков, управляющих потоком информации. Самым важным компонентом является блок состояния $s_i^{(t)}$ с линейной петлей, аналогичный описанным выше блокам с утечкой. Однако теперь вес петли (или ассоциированная временна́я константа) управляется **вентильным блоком забывания** $f_i^{(t)}$ (для временного шага t и ячейки i), который присваивает этому весу значение от 0 до 1 с помощью сигмоиды:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right), \tag{10.40}$$

где $\boldsymbol{x}^{(t)}$ – текущий входной вектор, $\boldsymbol{h}^{(t)}$ – вектор текущего скрытого слоя, содержащий выходы всех LSTM-ячеек, а $\boldsymbol{b}^f$, $\boldsymbol{U}^f$, $\boldsymbol{W}^f$ – соответственно смещения, веса входов и рекуррентные веса для вентилей забывания. Таким образом, внутреннее состояние

LSTM-ячейки обновляется по следующей формуле, в которой присутствует условный вес петли $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right), \tag{10.41}$$

где $\boldsymbol{b}$, $\boldsymbol{U}$ и $\boldsymbol{W}$ – соответственно смещения, веса входов и рекуррентные веса LSTM-ячейки. Блок **внешнего входного вентиля** $g_i^{(t)}$ вычисляется аналогично вентилю забывания (с использованием сигмоиды для получения значения от 0 до 1), но со своими параметрами:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right). \tag{10.42}$$

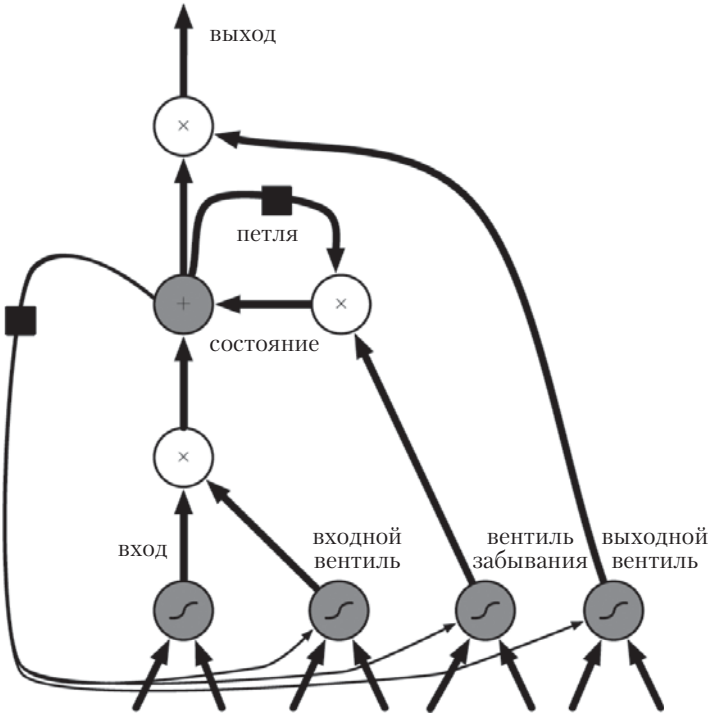

Рис. 10.16 ❖ Принципиальная схема «ячейки» рекуррентной LSTM-сети. Ячейки, рекуррентно связанные между собой, заменяют стандартные скрытые блоки в обыкновенных рекуррентных сетях. Входной признак вычисляется регулярным блоком искусственного нейрона. Его значение можно аккумулировать в состоянии, если сигмоидный входной вентиль это допускает. В блоке состояния имеется линейная петля, вес которой управляется вентилем забывания. Выход ячейки можно перекрыть с помощью выходного вентиля. Во всех вентильных блоках имеется сигмоидная нелинейность, тогда как во входном блоке разрешены произвольные сплющивающие нелинейности. Черным квадратиком обозначена задержка на одном временном шаге

Выход $h_i^{(t)}$ LSTM-ячейки можно перекрыть с помощью выходного вентиля $q_i^{(t)}$, в котором также используется сигмоида:

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)}, \tag{10.43}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right), \tag{10.44}$$

и $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$ – смещения, веса входов и рекуррентные веса соответственно. Один из вариантов – использовать состояние ячейки $s_i^{(t)}$ как дополнительный вход (со своим весом) всех трех вентилей i-го блока, как показано на рис. 10.16. Тогда потребуются три дополнительных параметра.

Показано, что LSTM-сети легче обучаются долгосрочным зависимостям, чем простые рекуррентные архитектуры: сначала на искусственных наборах данных, спроектированных специально для тестирования способности к обучению долгосрочным зависимостям (Bengio et al., 1994; Hochreiter and Schmidhuber, 1997; Hochreiter et al., 2001), а затем и на трудных задачах обработки последовательностей, при решении которых было достигнуто качество, не уступающее лучшим образцам (Graves, 2012; Graves et al., 2013; Sutskever et al., 2014). Ниже обсуждаются варианты и альтернативы LSTM, которые изучались и нашли практическое применение.

10.10.2. Другие вентильные РНС

Какие части архитектуры LSTM действительно необходимы? Какие еще можно придумать успешные архитектуры, позволяющие сети динамически управлять временным масштабом и поведением забывания различных блоков?

Ответы на некоторые из этих вопросов даны в недавних работах по вентильным РНС, блоки которых известны также под названием вентильных рекуррентных блоков (gated recurrent unit – GRU) (Cho et al., 2014b; Chung et al., 2014, 2015a; Jozefowicz et al., 2015; Chrupala et al., 2015). Основное отличие от LSTM заключается в том, что один вентильный блок одновременно управляет и коэффициентом забывания, и решением об обновлении блока состояния. Уравнения обновления имеют вид:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right), \tag{10.45}$$

где $\boldsymbol{u}$ обозначает вентиль «обновления», а $\boldsymbol{r}$ – вентиль «сброса». Их значения определяются как обычно:

$$u_i^{(t)} = \sigma\left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right); \tag{10.46}$$

$$r_i^{(t)} = \sigma\left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right). \tag{10.47}$$

Вентили обновления и сброса могут «игнорировать» части вектора состояния. Вентили обновления действуют как условные интеграторы с утечкой с линейной функцией по любому измерению, т. е. могут либо скопировать вход (один конец сигмоиды), либо полностью проигнорировать его (противоположный конец), заменив новым «целевым состоянием» (к которому интегратор с утечкой желает сойтись). Вентили сброса контролируют, какие части состояния использовать для вычисления

следующего целевого состояния, и вносят дополнительный нелинейный эффект в соотношение между прошлым и будущим состояниями.

На эту тему возможно еще много вариаций. Например, выход вентиля сброса (или вентиля забывания) можно разделить между несколькими скрытыми блоками. Или использовать произведение глобального вентиля (управляющего целой группой блоков, например всем слоем) и локального вентиля (управляющего одним блоком) для комбинирования глобального и локального управлений. Однако в нескольких исследованиях архитектурных вариантов LSTM и GRU не найдено решения, которое было бы очевидно лучше обоих на широком круге задач (Greff et al., 2015; Jozefowicz et al., 2015). В работе Greff et al. (2015) установлено, что вентиль забывания – ключевой ингредиент архитектуры, а в работе Jozefowicz et al. (2015) – что прибавление смещения 1 к вентилю забывания LSTM, рекомендованное в работе Gers et al. (2000), делает LSTM не уступающей лучшим из изученных архитектурных вариантов.

10.11. Оптимизация в контексте долгосрочных зависимостей

В разделах 8.2.5 и 10.7 описана проблема исчезающих и взрывных градиентов, возникающая при оптимизации РНС на большом числе временных шагов.

В работе Martens and Sutskever (2011) высказана интересная идея: вторые производные могут становиться исчезающе малыми одновременно с первыми. Алгоритмы оптимизации второго порядка можно грубо интерпретировать как деление первой производной на вторую (в многомерном случае – деление градиента на обратный гессиан). Если вторая производная убывает примерно с такой же скоростью, как первая, то отношение первой и второй производных будет оставаться относительно постоянным. К сожалению, у методов второго порядка много недостатков, в т. ч. высокая вычислительная стоимость, необходимость брать большой мини-пакет и притяжение к седловым точкам. В работе Martens and Sutskever (2011) получены многообещающие результаты с использованием методов второго порядка. Позже, в работе Sutskever et al. (2013), было установлено, что аналогичные результаты можно получить и более простыми методами, а именно методом Нестерова с тщательно подобранными начальными значениями. Дополнительные сведения см. в работе Sutskever (2012). Оба этих подхода в значительной степени заменены применением СГС (даже без импульса) к LSTM-сетям. Это пример постоянно встречающегося в машинном обучении явления: гораздо легче спроектировать простую для оптимизации модель, чем изобретать более мощный алгоритм оптимизации.

10.11.1. Отсечение градиентов

В разделе 8.2.4 отмечалось, что у сильно нелинейных функций, таких, например, как те, что вычисляются рекуррентной сетью через много временных шагов, производные нередко слишком малы или слишком велики по абсолютной величине. Это показано на рис. 8.3 и 10.17, где «ландшафт» целевой функции (как функции параметров) содержит «уступы»: широкие и довольно плоские участки, разделенные чрезвычайно узкими областями, где целевая функция быстро изменяется.

Трудность состоит в том, что когда градиент параметров очень велик, при обновлении параметров градиентного спуска параметры могут быть отброшены очень далеко в область бóльших значений целевой функции, и тогда значительная часть работы,

проделанной для нахождения текущего решения, пойдет насмарку. Градиент указывает направление, соответствующее наискорейшему спуску в бесконечно малой окрестности текущих параметров. Вне этой окрестности функция стоимости может снова начать возрастать. Обновление следует выбирать достаточно малым, чтобы предотвратить слишком сильный подъем. Обычно мы берем скорости обучения, снижающиеся достаточно медленно для того, чтобы на соседних шагах скорость обучения была примерно одинаковой. Размер шага, подходящий на сравнительно плоских участках ландшафта, зачастую не годится, т. к. приводит к подъему, когда на следующем шаге мы оказываемся на более искривленном участке.

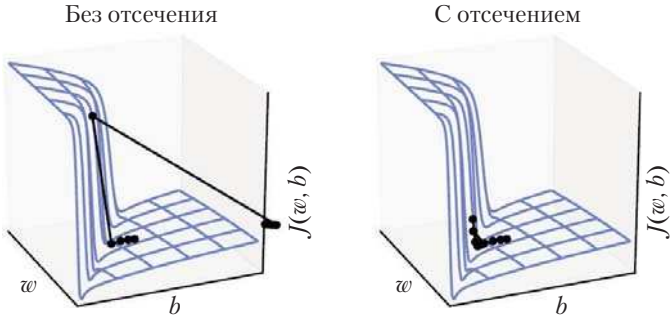

Рис. 10.17 ❖ Пример эффекта отсечения градиента в рекуррентной сети с двумя параметрами w и b. В результате отсечения метод градиентного спуска иногда ведет себя более разумно вблизи особенно крутых уступов. Такие крутые уступы часто встречаются в рекуррентных сетях рядом с участками, где сеть ведет себя почти линейно. Крутизна уступа экспоненциально возрастает с увеличением числа временных шагов, поскольку на каждом шаге матрица весов умножается на себя же. (*Слева*) Градиентный спуск без отсечения градиента проскакивает дно этого небольшого ущелья, после чего градиент резко возрастает на стенке уступа. Большой градиент приводит к катастрофическому уходу параметров от оптимума. (*Справа*) Градиентный спуск с отсечением градиента реагирует на уступ не так резко. Хотя подъем имеет место, размер шага ограничен, поэтому мы не можем уйти слишком далеко от крутой области рядом с решением. Рисунок взят из работы Pascanu et al. (2013) с разрешения авторов

На практике уже много лет используется простое решение: **отсечение градиента**. У этой идеи много вариантов (Mikolov, 2012; Pascanu et al., 2013). Один из них – отсекать градиент параметров на примерах из мини-пакета *поэлементно* (Mikolov, 2012), непосредственно перед обновлением параметров. Другой – *отсекать норму* $\|\boldsymbol{g}\|$ *градиента* $\boldsymbol{g}$ (Pascanu et al., 2013) непосредственно перед обновлением параметров:

$$\text{if } \|\boldsymbol{g}\| > v, \tag{10.48}$$

$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{\|\boldsymbol{g}\|}, \tag{10.49}$$

где v – порог нормы, а $\boldsymbol{g}$ используется для обновления параметров. Поскольку градиент всех параметров (включая различные группы, как, например, веса и смещения)

совместно перенормируется с одним масштабным коэффициентом, у второго метода есть преимущество: он гарантирует, что каждый шаг происходит все еще в направлении градиента; впрочем, эксперименты показывают, что оба метода работают примерно одинаково. Хотя обновление параметров направлено в ту же сторону, что истинный градиент, в случае отсечения градиента по норме норма вектора обновления оказывается ограниченной, вследствие чего алгоритм избегает убийственного шага при резком росте градиента. На самом деле даже случайный выбор шага в случае, когда абсолютная величина превышает порог, работает почти так же хорошо. Если рост настолько сильный, что градиент принимает значение `Inf` или `Nan` (бесконечность или «не число»), то, взяв случайный размер шага v, мы, как правило, уйдем из области численной неустойчивости. Отсечение нормы градиента на мини-пакет не изменяет направления градиента для отдельного мини-пакета. Однако усреднение отсеченных по норме градиентов по многим мини-пакетам не эквивалентно отсечению нормы истинного градиента (образованного при использовании всех примеров). Вклад примеров с большой нормой градиента, а также примеров, входящих в один мини-пакет с ними, в конечное направление уменьшается. Это контрастирует с традиционным мини-пакетным градиентным спуском, когда направление истинного градиента совпадает с результатом усреднения всех мини-пакетных градиентов. По-другому можно сказать, что в традиционном стохастическом градиентном спуске используется несмещенная оценка градиента, тогда как в градиентном спуске с отсечением по норме вводится эвристическое смещение, которое, как мы знаем, эмпирически полезно. В случае поэлементного отсечения направление обновления не совпадает с направлением истинного или мини-пакетного градиента, но все равно является приемлемым. Предлагалось (Graves, 2013) отсекать градиент на этапе обратного распространения (с учетом скрытых блоков), но результаты сравнения этого варианта с другими не опубликованы; отсюда мы заключаем, что все эти методы ведут себя более-менее одинаково.

10.11.2. Регуляризация с целью подталкивания информационного потока

Отсечение градиента помогает бороться с взрывными градиентами, но ничего не может поделать с исчезающими. Говоря о решении проблемы исчезающих градиентов и о лучшем улавливании долгосрочных зависимостей, мы обсуждали идею создания путей в графе вычислений развернутой рекуррентной архитектуры, вдоль которых произведение градиентов, ассоциированных с ребрами, близко к 1. Один из способов сделать это дают LSTM-сети и другие механизмы петель и вентилей, рассмотренные в разделе 10.10. Еще одна идея – регуляризировать параметры или наложить на них ограничения, с тем чтобы подтолкнуть «информационный поток». Конкретно, мы хотели бы, чтобы распространяющийся обратно вектор градиента $\nabla_{\boldsymbol{h}^{(t)}} L$ сохранял абсолютную величину, даже если функция потерь штрафует только выход в конце последовательности. Формально требуется, чтобы величина

$$(\nabla_{\boldsymbol{h}^{(t)}} L)\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \tag{10.50}$$

была так же велика, как

$$\nabla_{\boldsymbol{h}^{(t)}} L. \tag{10.51}$$

С этой целью в работе Pascanu et al. (2013) предложен такой регуляризатор:

$$\Omega = \sum_t \left(\frac{\left| \left| (\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \right| \right|}{\left\| \nabla_{\boldsymbol{h}^{(t)}} L \right\|} - 1 \right)^2. \tag{10.52}$$

Может показаться, что вычислить градиент этого регуляризатора трудно, но в работе Pascanu et al. (2013) предложена аппроксимация, в которой распространяющиеся обратно векторы $\nabla_{\boldsymbol{h}^{(t)}} L$ рассматриваются как постоянные (применительно к этому регуляризатору, т. е. нет необходимости производить через них обратное распространение). Эксперименты показывают, что в сочетании с эвристикой отсечения по норме (для борьбы с взрывными градиентами) этот регуляризатор может значительно увеличить временную протяженность зависимостей, которым может обучиться РНС. Поскольку он удерживает динамику РНС на грани взрывного роста градиентов, отсечение особенно важно – без него рост градиентов не позволит обучению успешно завершиться. Главный недостаток этого подхода в низкой, по сравнению с LSTM, эффективности на задачах, где данных в изобилии, например в задаче моделирования языка.

10.12. Явная память

Интеллект нуждается в знаниях, а знания приобретаются в процессе обучения, что и стало стимулом для разработки крупномасштабных глубоких архитектур. Однако знания бывают разные. Есть знания неявные, подсознательные, с трудом выражаемые словами: как ходить или чем собака отличается от кошки. Другие знания явные, декларативные, сравнительно легко облекаемые в слова – это могут быть как знания, основанные на бытовом здравом смысле, например «кошка – это животное», так и весьма специфичные факты, знать которые необходимо для достижения текущих целей, например «собрание отдела продаж состоится в 15:00 в комнате 141».

Нейронные сети прекрасно справляются с хранением неявных знаний, но сталкиваются с трудностями при запоминании фактов. Методу стохастического градиентного спуска нужно много раз предъявить одни и те же входные данные, чтобы он смог запомнить их в параметрах нейронной сети, но и тогда нельзя сказать, что эти данные хранятся точно. В работе Graves et al. (2014b) высказана гипотеза, что причина этого в том, что нейронным сетям недостает эквивалента **кратковременной памяти**, которая позволяет людям явно хранить и оперировать фрагментами информации, относящимися к достижению некоторой цели. Такие компоненты явной памяти позволили бы нашим системам не только быстро и «намеренно» запоминать и извлекать конкретные факты, но и последовательно рассуждать о них. Потребность в нейронных сетях, которые могут обрабатывать информацию, выполняя последовательность шагов, на каждом из которых изменяется способ подачи входных данных в сеть, давно уже осознана как важная предпосылка рассуждений, а не автоматической, интуитивной реакции на вход (Hinton, 1990).

Для разрешения этой проблемы в работе Weston et al. (2014) введена концепция **сетей с памятью**, в которые входят набор ячеек памяти и механизм их адресации. Оригинальные сети с памятью требовали инструкции от учителя по использованию ячеек памяти. В работе Graves et al. (2014b) введено понятие **нейронной машины Тьюринга**

(НМТ), способной обучиться чтению и записи произвольного содержимого в ячейки памяти без явных указаний со стороны учителя о том, какие действия предпринимать. Там же описан сквозной процесс обучения без сигнала от учителя посредством механизма мягкого внимания, основанного на содержимом (см. Bahdanau et al.[2015] и раздел 12.4.5.1). Этот механизм мягкой адресации стал стандартом и в других родственных архитектурах, он позволяет эмулировать алгоритмические механизмы, не отказываясь от градиентной оптимизации (Sukhbaatar et al., 2015; Joulin and Mikolov, 2015; Kumar et al., 2015; Vinyals et al., 2015a; Grefenstette et al., 2015).

Каждую ячейку памяти можно рассматривать как обобщение ячеек памяти в LSTM и GRU. Разница в том, что на выходе сети будет внутреннее состояние, которое выбирает, какую ячейку читать или записывать, точно так же, как при доступе к памяти компьютера задается адрес, по которому производить чтение или запись.

Трудно оптимизировать функции, порождающие точные целочисленные адреса. Чтобы как-то решить эту проблему, НМТ на самом деле читает и пишет сразу в несколько ячеек памяти. Результатом чтения является взвешенное среднее многих ячеек. А в процессе записи значения нескольких ячеек изменяются на разную величину. Коэффициенты для этих операций подбираются так, чтобы сфокусироваться на небольшом числе ячеек, например с применением функции softmax. Использование этих весов с ненулевыми производными позволяет применить метод градиентного спуска для оптимизации функций, управляющих доступом к памяти. Градиент по этим коэффициентам указывает, нужно ли их увеличить или уменьшить, но, как правило, градиент будет большим только для адресов памяти, получивших большой коэффициент.

Ячейки памяти обычно дополняются возможностью хранить вектор, а не только одиночный скаляр, как в ячейках LSTM или GRU. Есть две причины для увеличения размера ячейки памяти. Первая состоит в том, что мы увеличили стоимость доступа к ячейке. Мы платим за порождение коэффициента для многих ячеек, но ожидаем, что эти коэффициенты кластеризуются вокруг небольшого числа ячеек. Читая векторное, а не скалярное значение, мы амортизируем часть стоимости. Другая причина – то, что векторнозначные ячейки памяти допускают **адресацию по содержимому**, когда вес, используемый для чтения или записи ячейки, является функцией этой ячейки. Векторнозначные ячейки позволяют извлекать содержимое памяти целиком, если мы в состоянии сгенерировать образец, которому соответствуют некоторые, но не все элементы. Тут можно провести аналогию с тем, как человек вспоминает текст песни, услышав несколько слов. Инструкцию чтения на основе содержимого можно сравнить с указанием «извлечь слова песни, в которой есть припев “We all live in a yellow submarine”». Адресация по содержимому более полезна, когда извлекаемые объекты велики, – если бы каждая буква текста песни хранилась в отдельной ячейке памяти, мы не смогли бы найти их таким способом. Для сравнения – **позиционная адресация** не допускает ссылки на содержимое памяти. Инструкцию позиционной адресации можно было бы сформулировать так: «Извлечь слова песни по адресу 347». Позиционная адресация часто является вполне разумным механизмом, даже когда размер ячеек памяти мал.

Если содержимое ячейки памяти копируется (а не забывается) на большинстве временных шагов, то хранящуюся в ней информацию можно распространять вперед во времени, а градиенты – назад во времени, не опасаясь ни исчезновения, ни взрывного роста.

Явная память иллюстрируется на рис. 10.18, где «нейронная сеть для решения задачи» сопряжена с памятью. Нейронная сеть может быть как прямого распространения, так и рекуррентной, но система в целом является рекуррентной сетью. Сеть для решения задачи может выбирать, по каким адресам в памяти производить чтение или запись. Похоже, что явная память позволяет модели обучаться таким задачам, которые недоступны обыкновенным РНС или рекуррентным LSTM-сетям. Возможно, что одной из причин является тот факт, что информацию и градиенты можно распространять (соответственно вперед и назад во времени) в течение очень длительных промежутков времени.

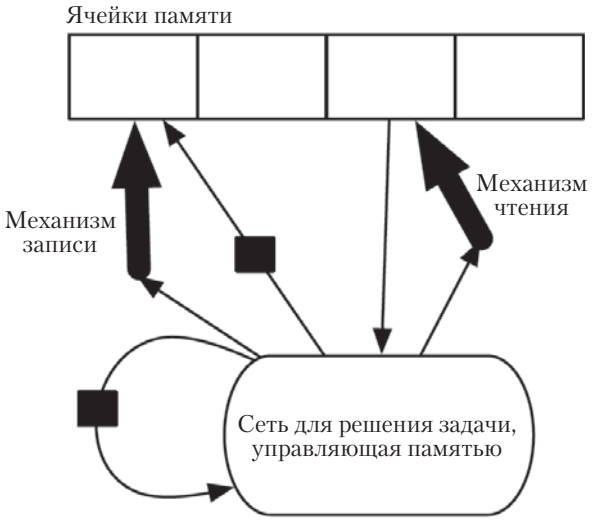

Рис. 10.18 ❖ Схема сети с явной памятью, отражающая некоторые ключевые элементы проекта нейронной машины Тьюринга. На этом рисунке «репрезентативная» часть модели («сеть для решения задачи», в данном случае рекуррентная сеть внизу) отделена от «памяти» (набора ячеек), в которой хранятся факты. Сеть для решения задачи обучается «управлять» памятью, т. е. решать, какие ячейки читать и записывать (с помощью механизмов чтения и записи, обозначенные жирными стрелками, которые указывают на адреса)

В качестве альтернативы обратному распространению посредством взвешенного среднего ячеек памяти мы можем интерпретировать коэффициенты адресации памяти как вероятности и стохастически читать только одну ячейку (Zaremba and Sutskever, 2015). Для оптимизации моделей, принимающих дискретные решения, нужны специальные алгоритмы, описанные в разделе 20.9.1. В настоящее время обучение стохастических архитектур, принимающих дискретные решения, остается более трудным делом, чем обучение детерминированных алгоритмов, принимающих мягкие решения.

Не важно, является ли механизм выбора адреса мягким (допускающим обратное распространение) или стохастическим и жестким, по своей форме он идентичен **механизму внимания**, который мы уже упоминали в контексте машинного перевода (Bahdanau et al., 2015) и будем подробнее обсуждать в разделе 12.4.5.1. Идея механиз-

мов внимания для нейронных сетей появилась даже раньше, в контексте генерации рукописных текстов (Graves, 2013), где этот механизм был ограничен только перемещением по последовательности вперед во времени. В случае машинного перевода и сетей с памятью фокус внимания на текущем и предыдущем шагах может находиться в совершенно разных местах.

Рекуррентные нейронные сети позволяют распространить глубокое обучение на последовательные данные. Это последний из основных инструментов глубокого обучения в нашем арсенале. Далее мы перейдем к вопросу о том, как выбирать подходящие инструменты при решении реальных задач.

Глава 11

Практическая методология

Для успешного применения глубокого обучения недостаточно просто хорошего знакомства с существующими алгоритмами и принципами их работы. Настоящий специалист должен еще знать, как выбрать алгоритм для конкретного приложения и как собрать и проанализировать полученные в ходе эксперимента данные, чтобы улучшить систему машинного обучения. В своей повседневной работе по созданию таких систем специалисту-практику приходится решать, нужно ли добавить еще данных, повысить или понизить емкость модели, добавить или убрать регуляризирующие признаки, стоит ли улучшить оптимизацию и приближенный вывод модели и как отладить ее программную реализацию. Все эти эксперименты занимают много времени, и это лишь самая меньшая из всех трудностей, поэтому так важно заранее выбрать правильный курс действий, а не тыкаться вслепую.

Основная часть этой книги посвящена различным моделям машинного обучения, алгоритмам обучения и целевым функциям. Поэтому может сложиться впечатление, будто самое важное качество специалиста по машинному обучению – знать как много больше разных методов и хорошо разбираться в математике. Но на практике успеха обычно добивается тот, кто правильно применяет широко известный алгоритм, а не прибегает к запутанному алгоритму, толком не понимая, как он работает. Для правильного применения алгоритма необходимо уверенно владеть какой-нибудь простой методологией. Многие рекомендации, приведенные в этой главе, заимствованы из работы Ng (2015).

Мы рекомендуем применять на практике следующий процесс проектирования.

- Определите цели – какую вы будете использовать метрику ошибок и с чем сравнивать результаты для оценки ошибки. Метрика ошибок должна определяться решаемой задачей.
- Как можно скорее организуйте комплексный рабочий конвейер, включающий и оценку показателей качества.
- Оснастите систему измерительными средствами для определения узких мест. Диагностируйте компоненты, работающие хуже, чем ожидалось, и разберитесь, чем вызвано плохое качество: переобучением, недообучением или дефектами данных либо программ.
- Основываясь на результатах измерений, шаг за шагом вносите изменения: сбор новых данных, подстройку гиперпараметров или смену алгоритмов.

В качестве примера мы будем использовать систему транскрипции уличных адресов Street View (Goodfellow et al., 2014d). Цель этого приложения – добавить здания

на карты Google Maps. Разъезжающие по городу автомобили Street View фотографируют здания и в каждой фотографии прикладывают GPS-координаты. Сверточная сеть распознает адрес дома на каждой фотографии, что позволяет Google Maps занести местонахождение соответствующего здания в базу данных. История разработки этого коммерческого приложения дает пример следования рекомендуемой методологии.

Переходим к описанию шагов процесса.

11.1. Показатели качества

Определение целей в терминах метрики ошибок – обязательный первый шаг, потому что от выбранной метрики ошибок зависят все последующие действия. Кроме того, вы должны понимать, какой уровень качества желателен.

Помните, что для большинства приложений невозможно достичь полной безошибочности. Байесовская частота ошибок дает минимальную частоту, на которую мы можем рассчитывать, даже если объем обучающих данных бесконечен и есть возможность восстановить истинное распределение вероятности. Причина может заключаться в том, что входные признаки содержат неполную информацию о выходной величине, или в том, что система принципиально стохастическая. Кроме того, нас ограничивает конечный объем обучающих данных.

Объем обучающих данных может быть ограничен по разным причинам. Если ваша цель – построить наилучший возможный продукт или услугу, то обычно можно собрать больше данных, но следует сопоставить выигрыш от дальнейшего уменьшения ошибки со стоимостью сбора дополнительных данных. Для сбора данных нужны время, деньги, а возможно, даже страдания людей (если, к примеру, для получения данных требуются инвазивные медицинские исследования). Если же цель – ответить на теоретический вопрос о том, какой алгоритм лучше работает на фиксированном эталонном тесте, то в спецификации теста обычно указан обучающий набор, и собирать дополнительные данные запрещено.

Как определить разумный уровень ожидаемого качества? В академических исследованиях обычно имеется некоторая оценка частоты ошибок, основанная на ранее опубликованных результатах эталонного тестирования. При разработке коммерческого продукта у нас есть представление о том, какая частота ошибок необходима, чтобы приложение можно было считать безопасным, рентабельным или привлекательным для пользователей. После того как реалистические требования к частоте ошибок сформулированы, в основе всех дальнейших решений должно быть стремление к достижению этой частоты.

Помимо целевого значения показателя качества, есть еще вопрос о выборе показателя. Для измерения эффективности полного приложения, включающего компонент машинного обучения, есть несколько показателей качества. Обычно это не то же самое, что функция стоимости, используемая при обучении модели. В разделе 5.1.2 отмечалось, что принято измерять верность, или, эквивалентно, частоту ошибок системы.

Однако во многих приложениях нужны более содержательные метрики.

Иногда одни ошибки обходятся гораздо дороже других. Например, в системе обнаружения почтового спама возможны ошибки двух типов: неправильная классификация нормального сообщения как спама и неправильный пропуск спама во входящую почту. Заблокировать нормальное сообщение гораздо хуже, чем пропустить сомнительное. Вместо того чтобы измерять частоту ошибок классификатора спама, хоте-

лось бы измерить некую полную стоимость, в которой стоимость блокировки нормальных сообщений выше стоимости пропуска спама.

Иногда требуется обучить бинарный классификатор, обнаруживающий редкие события, например тест для диагностики редкого заболевания. Предположим, что заболевание встречается у одного человека на миллион. Мы легко можем получить для этой задачи распознавания верность 99.9999, просто заставив классификатор всегда сообщать об отсутствии заболевания. Очевидно, что верность не годится для оценки качества такой системы. Решить проблему можно, если измерять не верность, а **точность** (precision) и **полноту** (recall). Точностью называется доля правильных ответов модели, а полнотой – доля обнаруженных истинных событий. Детектор, утверждающий, что нет ни одного больного, имеет идеальную точность, но нулевую полноту. Детектор, утверждающий, что больны все, достигает идеальной полноты, но его точность равна процентной доле людей, страдающих заболеванием (0.0001 в случае, когда заболеванием страдает один человек на миллион). При использовании точности и полноты часто строят ТП-кривую, когда по оси y откладывается точность, а по оси x – полнота. Порождаемая классификатором оценка выше, если подлежащее обнаружению событие действительно произошло. Например, сеть прямого распространения, предназначенная для обнаружения болезни, выводит оценку $\hat{y} = P(y = 1 \mid \boldsymbol{x})$ вероятности того, что человек, чьи медицинские анализы описываются признаками $\boldsymbol{x}$, болен. Сеть сообщает, что событие обнаружено, если эта оценка превышает заданный порог. Варьируя порог, мы можем отдавать предпочтение либо точности, либо полноте. Во многих случаях желательно охарактеризовать качество классификатора одним числом, а не кривой. Для этого точность p и полнота r объединяются в ***F*-меру**:

$$F = \frac{2pr}{p+r}. \tag{11.1}$$

Еще один вариант – сообщать площадь под ТП-кривой. В некоторых приложениях система машинного обучения может отказаться принимать решение. Это полезно, если алгоритм способен оценить степень уверенности в своем решении, особенно если неверное решение может нанести вред, а у оператора есть возможность взять ответственность на себя. Примером может служить система транскрипции Street View. Ее задача – найти на фотографии номер дома, чтобы можно было связать координаты места съемки с адресом на карте. Поскольку ценность неточной карты резко снижается, важно добавлять адрес, только если транскрипция правильна. Если система машинного обучения полагает, что правильность транскрипции сомнительна, то лучше всего перепоручить обработку фотографии человеку. Разумеется, система полезна, только если она кардинально уменьшает количество фотографий, требующих внимания человека. В этой ситуации естественным показателем качества является **покрытие**, т. е. доля примеров, для которых система смогла дать ответ. Необходимо соблюдать баланс между покрытием и верностью. Всегда можно получить стопроцентную верность, вообще отказавшись от обработки примеров, но тогда покрытие упадет до 0. В проекте Street View ставилась задача добиться верности транскрипции, не уступающей человеку, при 95-процентном покрытии. Считалось, что человек правильно транскрибирует адрес в 98% случаев.

Существует много других метрик. Например, можно измерять кликабельность, проводить опрос удовлетворенности пользователей и т. д. В специализированных приложениях применяются также специальные критерии.

Важно заранее определить, какой показатель качества мы будем улучшать, а затем сконцентрироваться на этой цели. Не имея четко сформулированных целей, трудно сказать, принесло некое изменение системы успех или нет.

11.2. Выбор базовой модели по умолчанию

После выбора целей и показателей качества следующим шагом разработки любого практического приложения является организация разумной комплексной системы, причем делать это надо как можно раньше. В этом разделе мы порекомендуем, с каких алгоритмов начинать в различных ситуациях. Помните, что прогресс в области глубокого обучения стремительный, поэтому вполне вероятно, что вскоре после выхода этой книги из печати наилучшими по умолчанию будут считаться другие алгоритмы.

В зависимости от сложности задачи вы, возможно, захотите на начальной стадии вообще обойтись без глубокого обучения. Если есть шанс, что задачу удастся решить, просто правильно подобрав несколько линейных весов, то лучше начать с простой статистической модели типа логистической регрессии.

Если заведомо понятно, что задача относится к «ИИ в чистом виде», как, например, распознавание объектов, распознавание речи, машинный перевод и т. д., то для начала стоит выбрать подходящую модель глубокого обучения.

Прежде всего выберите общую категорию моделей, исходя из структуры своих данных. Если вам нужно провести обучение с учителем с векторами фиксированного размера на входе, возьмите сеть прямого распространения с полносвязными слоями. Если известна топологическая структура входа (например, входом является изображение), то воспользуйтесь сверточной сетью. В этих случаях начинать следует с какого-нибудь кусочно-линейного блока (ReLU или его обобщения: Leaky ReLU, PreLu или maxout). Если входом или выходом является последовательность, то пользуйтесь вентильной рекуррентной сетью (LSTM или GRU).

В качестве алгоритма оптимизации стоит взять СГС с импульсом и затухающей скоростью обучения (к числу популярных схем затухания, которые на одних задачах работают лучше, на других хуже, относятся: линейное затухание до достижения фиксированной минимальной скорости обучения, экспоненциальное затухание и уменьшение скорости обучения в 2–10 раз при каждом выходе ошибки на контрольном наборе на плато). Еще одна разумная альтернатива – Adam. Пакетная нормировка может оказать заметное влияние на качество оптимизации, особенно в случае сверточных сетей и сетей с сигмоидными нелинейностями. Хотя на первой итерации имеет смысл опустить пакетную нормировку, к ней следует вернуться сразу, как только в ходе оптимизации появятся признаки проблем.

Если обучающий набор не насчитывает десятков миллионов примеров, следует с самого начала включить какие-то мягкие формы регуляризации. Ранняя остановка – почти универсальная рекомендация. Прореживание – отличный способ регуляризации, легко реализуемый и совместимый со многими моделями и алгоритмами обучения. Пакетная нормировка также иногда уменьшает ошибку обобщения и позволяет отказаться от прореживания благодаря шуму в оценке статистик, вносимому для нормировки каждой переменной.

Если ваша задача похожа на какую-то другую, уже хорошо изученную, то вы поступите мудро, если для начала скопируете модель и алгоритм, которые показали

наилучшие результаты раньше. Возможно, стоит даже скопировать уже обученную модель. Например, при решении разных задач компьютерного зрения нередко используются признаки из сверточной сети, обученной на наборе данных ImageNet (Girshick et al., 2015).

Часто задают вопрос, стоит ли начинать с обучения без учителя, описанного далее в части III. Все зависит от предметной области. Известно, что в некоторых задачах, скажем в обработке естественного языка, методы обучения без учителя, например погружение слов, приносят колоссальный эффект. А, к примеру, в компьютерном зрении современные методы обучения без учителя не дают особого выигрыша, за исключением обучения с частичным привлечением учителя, когда помеченных примеров очень мало (Kingma et al., 2014; Rasmus et al., 2015). Если ваше приложение относится к классу задач, для которых польза обучения без учителя доказана, включайте такую модель уже в первую версию комплексной системы. В противном случае поступайте так, только если решаемая задача принципиально не подразумевает учителя. Вы всегда сможете добавить обучение без учителя позже, если заметите, что начальная базовая модель переобучена.

11.3. Надо ли собирать дополнительные данные?

Построив начальный вариант комплексной системы, можно приступать к измерению качества алгоритма и поиску способов его улучшения. Часто начинающие испытывают соблазн что-то улучшить, пробуя много разных алгоритмов. Но обычно гораздо разумнее собрать больше данных, чем пытаться улучшить алгоритм обучения.

Как решить, следует ли собирать дополнительные данные? Прежде всего определите, устраивает ли вас качество на обучающем наборе. Если качество плохое, то алгоритм обучения не использует должным образом даже тех данных, что уже есть, так что собирать новые бессмысленно. Попробуйте вместо этого увеличить размер модели, добавив слои или скрытые блоки в каждый слой. Попробуйте также поиграть с алгоритмом обучения, например настройте гиперпараметр скорости обучения. Если модель большая и алгоритмы оптимизации тщательно настроены, а результата все равно нет, то, возможно, проблема в качестве обучающих данных. Быть может, они слишком сильно зашумлены или не содержат входов, необходимых для правильного предсказания выходов. Тогда стоит начать все сначала, собрав более чистые данные или используя улучшенный набор признаков.

Если качество на обучающем наборе приемлемое, то измерьте качество на тестовом наборе. Если и оно приемлемое, то все уже сделано. Если же качество на тестовом наборе гораздо хуже, чем на обучающем, то сбор дополнительных данных – одно из самых эффективных решений. Главное, что нужно принять во внимание, – стоимость и саму возможность сбора дополнительных данных, стоимость и возможность уменьшить ошибку тестирования другими способами, а также ожидаемый объем данных, при котором можно будет значительно повысить качество на тестовом наборе. Большие интернет-компании с миллионами и миллиардами пользователей могут собрать большие наборы данных, и стоимость этого мероприятия может оказаться намного ниже альтернатив, поэтому ответ почти всегда – увеличить объем обучающих данных. Например, разработка больших размеченных наборов данных была одним из главных факторов успеха при решении задачи о распознавании объектов. В других контекстах, например в медицинских приложениях, сбор дополнительных данных

может стоить дорого или вообще невозможен. Простая альтернатива сбору дополнительных данных – уменьшить размер модели или улучшить регуляризацию, добавив такие параметры, как коэффициенты снижения весов, или включив такие стратегии регуляризации, как прореживание. Если разрыв между качеством на обучающем и тестовом наборах остается неприемлемым даже после настройки гиперпараметров регуляризации, то рекомендуется собрать еще данные.

При решении вопроса о сборе дополнительных данных нужно еще определиться, сколько именно данных необходимо. Тут полезно построить кривые зависимости между размером обучающего набора и ошибкой обобщения, как на рис. 5.4. Экстраполяция таких кривых может подсказать, сколько еще данных необходимо для достижения требуемого уровня качества. Обычно незначительное увеличение общего числа примеров не оказывает заметного влияния на ошибку обобщения. Поэтому рекомендуется применять к размеру обучающего набора логарифмический масштаб, например удваивать число примеров в каждом новом эксперименте.

Если собрать намного больше данных невозможно, то остается только один способ уменьшить ошибку обобщения – улучшить сам алгоритм обучения. Но это уже научно-исследовательская работа, а не рекомендации для специалистов-прикладников.

11.4. Выбор гиперпараметров

Во многих алгоритмах глубокого обучения имеются гиперпараметры, управляющие различными аспектами поведения алгоритма. Одни влияют на время работы и потребление памяти, другие – на качество модели, восстанавливаемой в процессе обучения, и на ее способность давать правильные результаты при предъявлении новых примеров.

Есть два основных подхода к выбору гиперпараметров: ручной и автоматический. Для выбора вручную нужно понимать, для чего предназначен каждый гиперпараметр и как модель машинного обучения достигает хорошей обобщаемости. Алгоритмы автоматического выбора не требуют от пользователя понимания всех этих тонкостей, но зачастую обходятся гораздо дороже с вычислительной точки зрения.

11.4.1. Ручная настройка гиперпараметров

Для ручного задания гиперпараметров необходимо понимать связи между гиперпараметрами, ошибкой обучения, ошибкой обобщения и вычислительными ресурсами (памятью и временем работы), а следовательно, основательное знакомство с фундаментальными идеями, касающимися эффективной емкости алгоритма обучения, которые были изложены в главе 5.

Целью ручной настройки обычно является достижение наименьшей ошибки обобщения при ограничениях на время работы и потребление памяти. Мы не будем обсуждать, как различные гиперпараметры влияют на время работы и объем памяти, поскольку это сильно зависит от платформы.

Основная цель ручного подбора гиперпараметров – привести эффективную емкость модели в соответствие со сложностью задачи. Эффективная емкость ограничена тремя факторами: репрезентативной емкостью модели, способностью алгоритма обучения минимизировать функцию стоимости, используемую для обучения модели, и степенью регуляризации модели посредством функции стоимости и процедуры обучения. Чем больше число слоев модели и число скрытых блоков в каждом слое,

тем выше репрезентативная емкость, т. е. модель может представить более сложные функции. Но она не сможет обучиться всем этим функциям, если алгоритм обучения не способен установить, что некоторые функции хорошо минимизируют стоимость, или если регуляризирующие члены, например снижение весов, запрещают использование некоторых функций.

График ошибки обобщения как функции одного из гиперпараметров обычно имеет вид U-образной кривой (рис. 5.3). На одном конце значение гиперпараметра соответствует низкой емкости, а ошибка обобщения велика, потому что велика ошибка обучения. Это режим недообучения. На другом конце значение гиперпараметра соответствует высокой емкости, а ошибка обобщения велика, потому что велик разрыв между ошибкой обучения и тестирования. Где-то посередине находится оптимальная емкость модели, при которой ошибка обобщения минимальна, поскольку и разрыв, и ошибка обучения умеренные.

Для некоторых гиперпараметров слишком большое значение приводит к переобучению. Примером может служить число скрытых блоков в слое, поскольку с увеличением числа скрытых блоков возрастает емкость модели. А бывает и наоборот – когда к переобучению ведет малое значение гиперпараметра. Например, наименьшее допустимое значение коэффициента снижения весов, равное нулю, соответствует наибольшей эффективной емкости алгоритма обучения.

Не каждый гиперпараметр принимает все значения, лежащие на U-образной кривой. Многие гиперпараметры дискретны, как, например, число блоков в слое или число линейных участков maxout-блока, так что им соответствует лишь несколько точек на кривой. Бывают и бинарные гиперпараметры. Обычно это переключатели, которые говорят, нужно ли использовать какую-то необязательную компоненту алгоритма обучения, например шаг предобработки, на котором производится нормировка входных признаков (вычитание среднего и деление на стандартное отклонение). Таким гиперпараметрам соответствуют всего две точки на кривой. У некоторых гиперпараметров есть минимальное или максимальное значения, ограничивающие область их присутствия на кривой. Например, минимальный коэффициент снижения весов равен нулю. Это означает, что если модель недообучена, когда снижение весов нулевое, то путем модификации коэффициента снижения мы не сможем попасть в область переобучения. Иными словами, некоторые гиперпараметры способны только уменьшить емкость.

Пожалуй, самым важным гиперпараметром является скорость обучения. Если у вас есть время для настройки только одного гиперпараметра, займитесь скоростью обучения. Она управляет эффективной емкостью модели не столь прямо, как остальные гиперпараметры: эффективная емкость максимальна, когда скорость обучения *правильно* подобрана для задачи оптимизации, а не когда она особенно велика или мала. График зависимости ошибки *обучения* от скорости обучения имеет U-образную форму, показанную на рис. 11.1. Если скорость обучения слишком велика, то метод градиентного спуска может по случайности увеличить, а не уменьшить ошибку обучения. В идеализированном квадратичном случае это происходит, когда скорость обучения не менее чем в два раза превосходит оптимальное значение (LeCun et al., 1998a). Если скорость обучения слишком мала, то обучение не только протекает медленно, но может и вообще застрять, так и не добившись снижения ошибки обучения. Этот феномен пока понят плохо (но такое не может произойти для выпуклой функции потерь).

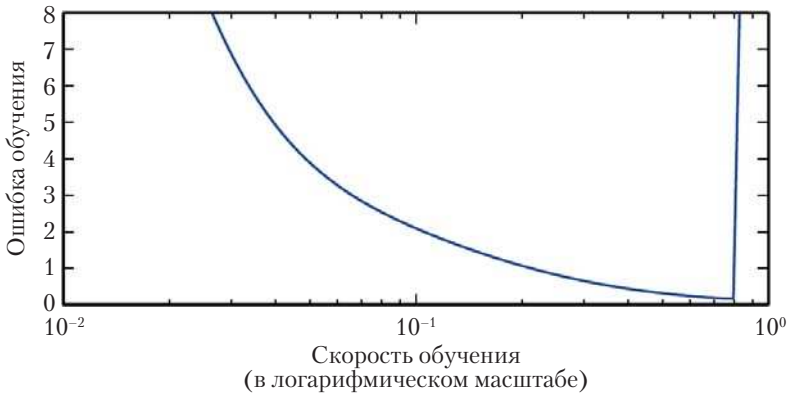

Рис. 11.1 ❖ Типичная зависимость ошибки обучения от скорости обучения. Обратите внимание на резкое возрастание ошибки, когда скорость выше оптимального значения. Это относится к фиксированному времени обучения, потому что снижение скорости обучения иногда может всего лишь замедлить обучение пропорционально уменьшению скорости. Ошибка обобщения может описываться этой же кривой или осложняется эффектами регуляризации, возникающими из-за слишком большой или слишком малой скорости обучения, поскольку неудачная оптимизация может до некоторой степени уменьшить или вообще предотвратить переобучение, и даже в точках с одинаковой ошибкой обучения ошибка обобщения может различаться

Для настройки параметров, отличных от скорости обучения, необходимо следить за ошибкой обучения и тестирования, чтобы понять, является ли модель переобученной или недообученной, а затем соответственно подкорректировать емкость.

Если ошибка на обучающем наборе выше целевой частоты ошибок, то нет другого выбора, кроме как увеличить емкость. Если вы не пользуетесь регуляризацией, но уверены в правильности алгоритма оптимизации, то следует увеличить число слоев сети или добавить скрытые блоки. К сожалению, это увеличивает вычислительную стоимость модели.

Если ошибка на тестовом наборе выше целевой частоты ошибок, то можно поступить двояко. Ошибка тестирования равна сумме ошибки обучения и разрыва между ошибкой тестирования и ошибкой обучения. Для нахождения оптимальной ошибки тестирования нужно сбалансировать эти величины. Нейронные сети обычно работают оптимально, когда ошибка обучения очень мала (а потому емкость велика), а ошибка тестирования в основном обусловлена разрывом между ошибкой обучения и тестирования. Ваша цель – уменьшить этот разрыв, не увеличивая ошибку обучения быстрее, чем сокращается разрыв. Для уменьшения разрыва измените гиперпараметры регуляризации, так чтобы уменьшить эффективную емкость модели, например добавьте прореживание или снижение весов. Обычно наилучшее качество достигается для большой хорошо регуляризированной (например, с помощью прореживания) модели.

Большинство гиперпараметров можно задать, поняв, увеличивают они емкость модели или уменьшают. В табл. 11.1 приведено несколько примеров.

Таблица 11.1. Влияние различных гиперпараметров на емкость модели

Гипер-параметр	Увеличивает емкость, когда…	Причина	Подводные камни
Число скрытых блоков	Возрастает	Увеличение числа скрытых блоков увеличивает репрезентативную емкость модели	Увеличение числа скрытых блоков приводит к увеличению времени работы и объема потребляемой памяти для всех операций модели
Скорость обучения	Настроена оптимально	Если скорость обучения слишком велика или слишком мала, эффективная емкость модели падает из-за плохой оптимизации	
Ширина ядра свертки	Возрастает	Чем больше ширина ядра, тем больше параметров у модели	Увеличение ширины ядра приводит к уменьшению ширины выхода, а значит, и к уменьшению емкости модели, если только не применяется дополнение нулями для компенсации этого эффекта. Чем шире ядро, тем больше памяти нужно для хранения параметров и тем больше время работы, однако уменьшение ширины выхода снижает потребление памяти
Неявное дополнение нулями	Возрастает	Добавление неявных нулей перед сверткой позволяет сохранить объем представления	Увеличивается время работы и потребление памяти
Коэффициент снижения весов	Убывает	Благодаря уменьшению коэффициента снижения весов открывается возможность для увеличения параметров модели	
Коэффициент прореживания	Убывает	Не столь частое выбрасывание блоков дает блокам возможность «договориться» между собой об аппроксимации обучающего набора	

Увлекшись ручной настройкой гиперпараметров, не упустите из виду конечную цель: хорошее качество на тестовом наборе. Один из путей достижения этой цели – добавление регуляризации. При условии что ошибка обучения остается малой, вы всегда можете уменьшить ошибку обобщения, собрав дополнительные обучающие данные. Лобовой способ, практически гарантирующий успех, – продолжать увеличивать емкость модели и размер обучающего набора, пока задача не будет решена. Конечно, при этом растет вычислительная стоимость обучения и вывода, так что применять этот способ можно только при наличии достаточных ресурсов. В принципе, этот подход может потерпеть неудачу из-за трудностей оптимизации, но для многих задач оптимизация не является серьезным барьером, если только выбрана подходящая модель.

11.4.2. Алгоритмы автоматической оптимизации гиперпараметров

Идеальный алгоритм обучения принимает набор данных и выводит функцию, не требуя ручной настройки гиперпараметров. Популярность таких алгоритмов, как ло-

гистическая регрессия и метод опорных векторов, зиждется на способности хорошо работать после настройки всего одного-двух гиперпараметров. Нейронные сети иногда тоже показывают неплохие результаты при небольшом числе гиперпараметров, но чаще настраивать приходится сорок, а то и больше. Ручная настройка гиперпараметров может дать отличный эффект, если у пользователя есть хорошая отправная точка, например определенная другими исследователями, работавшими с аналогичным приложением и архитектурой, или если пользователь имеет многомесячный или многолетний опыт изучения оптимальных значений гиперпараметров для подобных нейронных сетей. Увы, для многих приложений такой отправной точки нет. И тогда к отысканию полезных значений гиперпараметров можно привлечь автоматизированные алгоритмы.

Поразмыслив о том, как пользователь алгоритма обучения ищет хорошие значения гиперпараметров, мы придем к выводу о наличии оптимизации: мы пытаемся найти такие значения, которые оптимизируют некоторую целевую функцию, например ошибку на контрольном наборе, иногда в условиях ограничений (на время обучения, на потребление памяти или на время распознавания). Поэтому, в принципе, возможно разработать алгоритмы **оптимизации гиперпараметров**, которые обертывают алгоритм обучения и подбирают для него гиперпараметры, скрывая их от пользователя. К сожалению, у алгоритмов оптимизации гиперпараметров есть свои гиперпараметры, например диапазон потенциальных значений каждого из гиперпараметров алгоритма обучения. Но такие вторичные гиперпараметры задать обычно проще в том смысле, что один и тот же их набор позволяет добиться приемлемого качества для широкого круга задач.

11.4.3. Поиск на сетке

Когда гиперпараметров всего три или того меньше, часто прибегают к **поиску на сетке**. Для каждого гиперпараметра пользователь выбирает небольшое конечное множество потенциальных значений. Затем алгоритм поиска обучает модель для каждой возможной комбинации значений гиперпараметров. Наилучшими считаются гиперпараметры, при которых была достигнута наименьшая ошибка на контрольном наборе. На рис. 11.2 слева показана сетка значений гиперпараметров.

Как выбирать списки потенциальных значений? В случае числовых (упорядоченных) гиперпараметров наибольший и наименьший элементы каждого списка выбираются с запасом, основываясь на результатах аналогичных экспериментов в прошлом, чтобы оптимальное значение с большой вероятностью оказалось в выбранном диапазоне. Как правило, при поиске на сетке значения выбираются примерно в логарифмическом масштабе, например скорость обучения берется из множества $\{0.1, 0.01, 10^{-3}, 10^{-4}, 10^{-5}\}$, а число скрытых блоков – из множества $\{50, 100, 200, 500, 1000, 2000\}$.

Поиск на сетке обычно дает наилучшие результаты, если повторяется несколько раз. Предположим, к примеру, что мы провели поиск на сетке гиперпараметра α с потенциальными значениями $\{-1, 0, 1\}$. Если наилучшим оказалось значение 1, то мы не полностью рассмотрели диапазон, в котором находится оптимальное значение α, и потому должны повторить поиск на сдвинутой сетке, например $\{1, 2, 3\}$. Если же наилучшим оказалось значение 0, то следует уточнить оценку, уменьшив масштаб, например, на сетке $\{-0.1, 0, 0.1\}$.

С поиском на сетке связана очевидная проблема: вычислительная стоимость экспоненциально растет с увеличением числа гиперпараметров. Если имеется m гипер-

параметров и каждый принимает n значений, то число раундов обучения-вычисления растет как $O(n^m)$. Раунды можно распараллелить, причем между машинами, на которых производятся вычисления, почти не требуется передавать данных. К сожалению, из-за экспоненциальной стоимости поиска на сетке даже распараллеливание может не принести желаемых результатов.

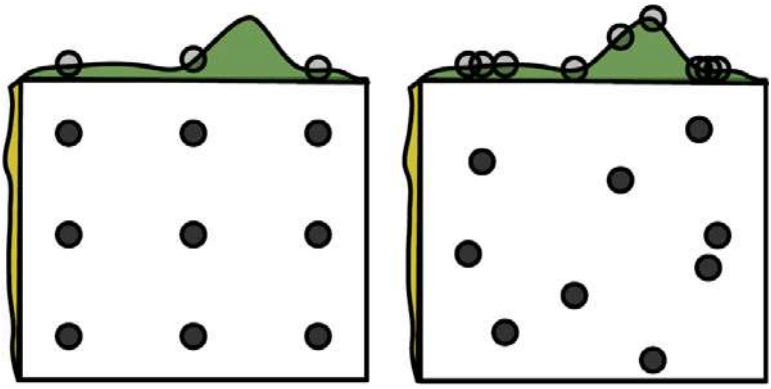

Рис. 11.2 ❖ Сравнение поиска на сетке и случайного поиска. Для наглядности показаны всего два гиперпараметра, но на практике их обычно гораздо больше. (*Слева*) Для поиска на сетке задается множество значений каждого гиперпараметра. Алгоритм производит обучение для каждой комбинации этих значений. (*Справа*) Для случайного поиска задается распределение вероятности совместных конфигураций гиперпараметров. Как правило, гиперпараметры независимы друг от друга. В качестве распределения одного гиперпараметра часто берут равномерное или логарифмически равномерное (для выборки из логарифмически равномерного распределения нужно взять экспоненту значения, выбранного из равномерного распределения). Алгоритм поиска производит случайную выборку из совместного распределения гиперпараметров и выполняет обучение для каждой выбранной комбинации. Оба алгоритма – поиска на сетке и случайного поиска – вычисляют ошибку на контрольном наборе и возвращают лучшую из найденных конфигураций. На рисунке показан типичный случай, когда лишь некоторые гиперпараметры оказывают существенное влияние на результат. В данном случае существенным оказался только гиперпараметр, отложенный по горизонтальной оси. Время, впустую расходуемое алгоритмом поиска на сетке, экспоненциально зависит от числа несущественных гиперпараметров, тогда как алгоритм случайного поиска почти на каждом испытании проверяет уникальные значения каждого существенного гиперпараметра. Рисунок взят из работы Bergstra and Bengio (2012) с разрешения авторов

11.4.4. Случайный поиск

По счастью, поиску на сетке есть альтернатива, которая столь же проста для программирования, но удобнее в использовании и сходится к хорошим значениям гиперпараметров гораздо быстрее: случайный поиск (Bergstra and Bengio, 2012).

Случайный поиск производится следующим образом. Сначала задаем маргинальное распределение каждого гиперпараметра, например распределение Бернулли и ка-

тегориальное распределение для бинарных и дискретных параметров соответственно либо логарифмически равномерное распределение для гиперпараметров, принимающих положительные вещественные значения. Например:

$$\texttt{log\_learning\_rate} \sim u(-1, -5), \tag{11.2}$$

$$\texttt{learning\_rate} = 10^{\texttt{log\_learning\_rate}}, \tag{11.3}$$

где $u(a, b)$ – выборка из равномерного распределения в интервале (a, b). Аналогично `log_number_of_hidden_units` можно выбрать из $u(\log(50), \log(2000))$.

В отличие от поиска на сетке, здесь не нужно дискретизировать значения гиперпараметров, поэтому можно исследовать более широкое множество значений, избежав дополнительных вычислительных затрат. Как показано на рис. 11.2, случайный поиск может оказаться экспоненциально эффективнее поиска на сетке, если существует несколько гиперпараметров, не оказывающих существенного влияния на показатель качества. Этот вопрос скрупулезно изучен в работе Bergstra and Bengio (2012), где показано, что случайный поиск снижает ошибку на контрольном наборе гораздо быстрее, чем поиск на сетке, если оценивать по числу раундов.

Как и при поиске на сетке, зачастую имеет смысл повторить случайный поиск, чтобы уточнить значения, найденные при первом прогоне.

Основная причина, по которой случайный поиск находит хорошие решения быстрее, чем поиск на сетке, – отсутствие лишних экспериментальных раундов, встречающихся при поиске на сетке, когда два значения некоторого гиперпараметра (при заданных значениях остальных) дают один и тот же результат. При поиске на сетке у остальных значений гиперпараметра в этих двух раундах будут одинаковые значения, а при случайном поиске – обычно разные. Поэтому если различие между этими двумя значениями не оказывает существенного влияния на ошибку на контрольном наборе, то поиск на сетке без нужды повторяет два эквивалентных эксперимента, тогда как случайный поиск произведет два независимых испытания других гиперпараметров.

11.4.5. Оптимизация гиперпараметров на основе модели

Поиск хороших значений гиперпараметров можно рассматривать как задачу оптимизации. Параметрами решения являются гиперпараметры. Подлежащая оптимизации функция стоимости – ошибка на контрольном наборе, возникающая при обучении с такими гиперпараметрами. В упрощенной постановке, когда можно вычислить градиент какой-то дифференцируемой метрики ошибки на контрольном наборе относительно гиперпараметров, мы можем просто двигаться в направлении этого градиента (Bengio et al., 1999; Bengio, 2000; Maclaurin et al., 2015). К сожалению, в большинстве практических задач этот градиент недоступен – либо из-за высокой стоимости вычислений и потребности в памяти, либо потому что взаимодействие гиперпараметров с ошибкой на контрольном наборе описывается недифференцируемой функцией, как в случае дискретных гиперпараметров.

Чтобы компенсировать отсутствие градиента, мы можем построить модель ошибки на контрольном наборе, а затем выдвинуть гипотезу о значениях гиперпараметров, выполнив оптимизацию в рамках этой модели. В большинстве подобных алгоритмов поиска гиперпараметров применяется байесовская модель регрессии для оценки как ожидаемого значения ошибки для каждого гиперпараметра, так и неопределенности

этой оценки. Таким образом, оптимизация подразумевает поиск компромисса между исследованием (предложением гиперпараметров с высокой неопределенностью, которые могут дать заметное улучшение, а могут и не дать) и использованием (предложением гиперпараметров, относительно которых модель уверена, что они будут работать так же хорошо, как виденные ей ранее, – обычно это значения, очень близкие к наблюдавшимся прежде). Из современных подходов к оптимизации гиперпараметров отметим алгоритмы Spearmint (Snoek et al., 2012), TPE (Bergstra et al., 2011) и SMAC (Hutter et al., 2011).

В настоящее время мы не можем однозначно порекомендовать байесовскую оптимизацию гиперпараметров в качестве общепринятого средства улучшения результатов глубокого обучения или получения результатов с меньшими усилиями. Иногда байесовская оптимизация дает результаты, сравнимые с полученными человеком, иногда даже лучшие, но терпит катастрофические неудачи на других задачах. Посмотреть, как она будет работать на конкретной задаче, пожалуй, имеет смысл, но пока этот подход нельзя назвать ни зрелым, ни надежным. Тем не менее оптимизация гиперпараметров – важная область исследований, и хотя ее развитием движут в основном потребности глубокого обучения, ее достижения сулят выигрыш не только машинному обучению в целом, но и всем техническим дисциплинам.

Всем алгоритмам оптимизации гиперпараметров, более сложным, чем случайный поиск, присущ общий недостаток: чтобы из раунда обучения можно было извлечь полезную информацию, он должен быть доведен до конца. Это сильно снижает эффективность, поскольку обычно уже на ранних стадиях эксперимента человек может сказать, что некоторая комбинация гиперпараметров абсолютно бессмысленна. В работе Swersky et al. (2014) приведена первая версия алгоритма, который хранит набор из нескольких экспериментов. В различные моменты времени алгоритм оптимизации гиперпараметров может принять решение: начать новый эксперимент, «заморозить» текущий эксперимент, не обещающий ничего интересного, или «разморозить» и возобновить эксперимент, который ранее был заморожен, но теперь, с появлением новой информации, выглядит многообещающе.

11.5. Стратегии отладки

Если система машинного обучения работает плохо, то обычно трудно сказать, в чем корень зла: то ли это недостаток самого алгоритма, то ли ошибка в его реализации. Отладка систем машинного обучения затруднена по разным причинам.

В большинстве случаев нам заранее неизвестно предполагаемое поведение алгоритма. Ведь суть машинного обучения в том и состоит, чтобы выявить некое полезное поведение, которое мы не смогли описать сами. Если мы обучили нейронную сеть *новой* задаче классификации и получили на тестовом наборе ошибку 5 процентов, то как узнать, ожидаемое это поведение или далеко не оптимальное?

Следующая трудность заключается в том, что у большинства моделей машинного обучения несколько частей, и все они адаптивны. Если какая-то часть работает неправильно, то остальные могут адаптироваться, и общее качество останется более-менее приемлемым. Предположим, к примеру, что мы обучаем нейронную сеть с несколькими слоями, параметризованными весами $\boldsymbol{W}$ и смещениями $\boldsymbol{b}$. Предположим еще, что мы вручную реализовали правила градиентного спуска для каждого параметра в отдельности, но допустили ошибку в обновлении смещений:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha, \tag{11.4}$$

где α – скорость обучения. В этом ошибочном правиле обновления градиент не используется вовсе. Поэтому в ходе обучения смещения всегда будут становиться отрицательными, что, очевидно, не может считаться корректной реализацией разумного алгоритма обучения. Но найти эту ошибку путем изучения выхода модели не всегда легко. При некоторых распределениях входных данных веса могут адаптироваться и компенсировать отрицательные смещения.

Большинство стратегий отладки нейронных сетей предназначено для преодоления одной из этих трудностей или сразу обеих. Либо мы строим тест настолько простой, что правильное поведение можно предсказать, либо такой, в котором проверяется лишь одна часть реализации сети изолированно от остальных.

Перечислим несколько важных стратегий отладки.

Визуализировать модель в действии. Во время обучения модели обнаружения объектов в изображении показать несколько изображений с наложенными контурами обнаруженных объектов. Если обучается порождающая модель речи, прослушать несколько сгенерированных ей образчиков. Вроде бы совершенно очевидно, но очень легко впасть в грех ознакомления только с количественными измерениями качества работы, например верности или логарифмического правдоподобия. Прямое наблюдение за тем, как модель справляется со своей задачей, помогает установить, являются ли достигнутые ей количественные показатели разумными. Ошибки оценивания могут быть самыми губительными, потому что вселяют уверенность, что система работает правильно, хотя на самом деле это совсем не так.

Визуализация самых серьезных ошибок. Большинство моделей может вывести какой-то показатель уверенности в полученных результатах. Например, классификаторы, в которых выходной слой основан на функции softmax, назначают вероятность каждому классу. Следовательно, вероятность, назначенная самому вероятному классу, и дает оценку уверенности модели в принятом решении. Обычно обучение на основе максимального правдоподобия дает завышенные оценки, а не точные вероятности правильного предсказания, но все равно они полезны, поскольку примеры, которые с меньшей вероятностью помечены правильно, получат от модели меньшую оценку вероятности. Изучив примеры, для которых модель испытывает наибольшие трудности, зачастую удается вскрыть проблемы в предобработке или пометке данных. Например, в системе транскрипции Street View была ошибка, из-за которой система обнаружения номера дома слишком сильно обрезала изображение, так что несколько цифр выпадало из кадра. Затем сеть транскрипции назначала слишком низкую вероятность правильному ответу для таких изображений. После того как ошибка была исправлена и ширина обрезанного кадра стало гораздо больше, качество системы в целом резко возросло, хотя сети пришлось иметь дело с большей вариативностью позиций и масштабов номеров домов.

Логическое рассуждение о программе с использованием ошибок обучения и тестирования. Часто бывает трудно определить, правильно ли реализована программа. Кое-какую информацию можно получить из анализа ошибок обучения и тестирования. Если ошибка обучения мала, а ошибка тестирования велика, то, вероятно, процедура обучения работает правильно, а модель переобучена вследствие фундаментальных алгоритмических причин. Возможно также, что ошибка тестирования измерена неправильно из-за проблем в связи с сохранением обученной модели и последующей

ее загрузкой для обработки тестового набора или из-за того, что тестовые данные готовились не так, как обучающие. Если и ошибка обучения, и ошибка тестирования велики, то трудно сказать, то ли это дефект программы, то ли модель недообучена вследствие фундаментальных алгоритмических причин. В таком случае необходимы дополнительные тесты, описанные ниже.

Аппроксимация крохотного набора данных. Если ошибка на обучающем наборе велика, нужно решить, в чем причина: в недообученности или в дефекте программы. Обычно даже небольшую модель можно гарантированно обучить аппроксимации достаточно малого набора. Например, если набор для обучения классификатора содержит всего один пример, то для его аппроксимации достаточно просто правильно установить смещения в выходном слое. Если вы не можете обучить классификатор правильно помечать один пример, автокодировщик – успешно воспроизводить один пример с высокой точностью, или порождающую модель – устойчиво генерировать примеры, напоминающие заданный, то следует заподозрить ошибку в программе, препятствующую успешной оптимизации на обучающем наборе. Этот тест можно обобщить на наборы данных с небольшим числом примеров.

Сравнение производных, вычисленных методом обратного распространения, с численными производными. Если программная система требует, чтобы вы самостоятельно реализовали вычисления градиента, или если вы добавили новую операцию в библиотеку дифференцирования и должны определить для нее метод `bprop`, то частой причиной ошибок является некорректная реализация выражения градиента. Проверить, так ли это, можно, сравнив производные, вычисленные вашей процедурой автоматического дифференцирования, с производными, вычисленными методом конечных разностей. Поскольку

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}, \tag{11.5}$$

мы можем аппроксимировать производную, взяв малое значение ε:

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}. \tag{11.6}$$

Точность аппроксимации можно повысить, вычислив **центральную разность**:

$$f'(x) \approx \frac{f\left(x + \frac{1}{2}\varepsilon\right) - f\left(x - \frac{1}{2}\varepsilon\right)}{\varepsilon}. \tag{11.7}$$

Величина возмущения ε должна быть достаточно большой, чтобы предотвратить слишком сильное округление в процессе вычисления конечных разностей.

Обычно мы хотим протестировать градиент якобиана или векторной функции $g : \mathbb{R}^m \longrightarrow \mathbb{R}^n$. К сожалению, метод конечных разностей позволяет вычислять только одну производную в каждый момент времени. Следовательно, можно либо применить его mn раз для вычисления всех частных производных g, либо выполнить тест для новой функции, в которой берутся случайные проекции на вход и выход g. Например, можно применить тест к реализации производных функции $f(x) = \boldsymbol{u}^T g(\boldsymbol{v}x)$, где $\boldsymbol{u}$ и $\boldsymbol{v}$ – случайно выбранные векторы. Для правильного вычисления $f'(x)$ нужно уметь корректно выполнять обратное распространение через g, но это можно сделать эффективно и с помощью метода конечных разностей, потому что у f всего один ска-

лярный аргумент и одно выходное значение. Обычно имеет смысл повторить тест для нескольких значений $\boldsymbol{u}$ и $\boldsymbol{v}$, чтобы не проглядеть ошибок, ортогональных направлению случайного проецирования.

Если речь идет о вычислениях с комплексными числами, то существует очень эффективный способ численной оценки градиента путем передачи функции комплексного аргумента (Squire and Trapp, 1998). В основе метода лежит следующее наблюдение:

$$f(x + i\varepsilon) = f(x) + i\varepsilon f'(x) + O(\varepsilon^2), \tag{11.8}$$

$$\mathrm{Re}(f(x + i\varepsilon)) = f(x) + O(\varepsilon^2),\ \mathrm{Im}\left(\frac{f(x+i\varepsilon)}{\varepsilon}\right) = f'(x) + O(\varepsilon)^2. \tag{11.9}$$

где $i = \sqrt{-1}$. В отличие от рассмотренного выше вещественного случая, здесь нет потери значащих цифр, потому что вычисляется разность между значениями f в разных точках. Это позволяет брать очень малые значения ε, например 10^{-150}, так что погрешность $O(\varepsilon^2)$ практически несущественна.

Мониторинг гистограмм активаций и градиента. Часто бывает полезно визуализировать статистику активаций и градиента нейронной сети, собранную по многим итерациям обучения (быть может, по одному периоду). Значения скрытых блоков до активации могут сказать, являются ли блоки насыщенными и как часто такое случается. Например, если речь идет о блоках линейной ректификации, то как часто они выключены? А есть ли блоки, которые всегда выключены? В случае tanh-блоков о насыщении блока говорит среднее абсолютных величин до активации. В глубокой сети, где распространяемые градиенты быстро растут или быстро приближаются к нулю, могут возникнуть препятствия для оптимизации. Наконец, полезно сравнить модули градиентов параметров с модулями самих параметров. В работе Bottou (2015) рекомендуется, чтобы модуль изменения параметра по мини-пакету составлял примерно 1% от модуля параметра, а не 50% и не 0.001% (тогда изменение параметров было бы слишком медленным). Может оказаться, что некоторые группы параметров изменяются в хорошем темпе, тогда как другие топчутся на месте. Если данные разрежены (как в естественном языке), то некоторые параметры могут обновляться очень редко, и об этом следует помнить, наблюдая за их эволюцией.

Наконец, многие алгоритмы глубокого обучения дают некоторые гарантии относительно результатов на каждом шаге. Например, в части III мы встретимся с алгоритмами приближенного вывода, в основе которых лежит алгебраическое решение задач оптимизации. Обычно для их отладки можно проверить выполнение каждой гарантии. Приведем несколько примеров подобных гарантий: целевая функция никогда не увеличивается после одного шага алгоритма; градиент относительно некоторого подмножества переменных равен 0 после каждого шага алгоритма; градиент по всем переменным равен 0 по достижении сходимости. Обычно из-за ошибок округления эти условия выполняются неточно, поэтому при отладке нужно делать небольшой допуск.

11.6. Пример: распознавание нескольких цифр

В качестве примера практического применения комплексной методологии проектирования мы кратко опишем систему транскрипции Street View с точки зрения проектирования компонентов глубокого обучения. Понятно, что не менее важную роль играют и многие другие компоненты системы, в частности автомобили Street View,

инфраструктура базы данных и т. д. С точки зрения машинного обучения процесс начинается со сбора данных. Автомобили собирают исходные данные, а люди помечают их. Собственно, транскрипции предшествовала большая работа по подготовке набора данных, в т. ч. применение других методов машинного обучения для обнаружения номеров данных.

В начале работы над проектом мы определили показатели качества и их желательные значения. Важный общий принцип – выбирать показатели в соответствии с бизнес-целями проекта. Поскольку карты ценны, лишь если они верны, важным требованием к проекту было достижение высокой верности. Точнее, ставилась цель достичь такой же верности, как при определении номеров домов человеком, – 98%. Добиться такой верности не всегда возможно, поэтому система транскрипции Street View пожертвовала покрытием. Таким образом, покрытие стало главным показателем качества, который и требовалось оптимизировать при ограничении на верность 98%. По мере совершенствования нейронной сети появилась возможность снизить порог доверия, ниже которого система отказывалась транскрибировать вход, так что в итоговой системе покрытие превзошло намеченные 95%.

Следующим после выбора количественных целей шагом рекомендуемой методологии является скорейшая организация разумной базовой системы. Для задач компьютерного зрения это сверточная сеть с блоками линейной ректификации. Именно такую модель мы и выбрали в начале проекта. В то время редко можно было встретить сверточные сети, порождавшие последовательность предсказаний. В первой, самой простой модели выходной слой состоял из n softmax-блоков для предсказания последовательности n символов. Эти блоки обучались так, как если бы речь шла о задаче классификации, каждый softmax-блок обучался независимо от остальных.

Наша методология рекомендует итеративно уточнять базовую модель и проверять, привело ли очередное изменение к улучшению. Первое изменение системы транскрипции Street View было связано с теоретическим осмыслением метрики покрытия и структуры данных. Точнее, система отказывалась классифицировать вход $\boldsymbol{x}$, если вероятность выходной последовательности $p(\boldsymbol{y} \mid \boldsymbol{x}) < t$ для некоторого порога t. Первоначальное определение $p(\boldsymbol{y} \mid \boldsymbol{x})$ было упрощенным, основанным на простом перемножении всех выходов функций softmax. Возникла потребность в разработке специализированного выходного слоя и функции стоимости, которая вычисляла бы теоретически правильное логарифмическое правдоподобие. При таком подходе механизм отклонения примеров стал работать намного эффективнее.

На этом этапе покрытие оставалось ниже 90%, но очевидных теоретических проблем не просматривалось. Поэтому в соответствии с нашей методологией нужно было оснастить систему средствами измерения качества на обучающем и тестовом наборах, чтобы понять, вызвана проблема переобучением или недообучением. В данном случае ошибки на обучающем и тестовом наборах были почти одинаковы. Вообще, главная причина, по которой работа над проектом шла так гладко, – доступность набора данных, содержащего десятки миллионов помеченных примеров. Коль скоро ошибки на обоих наборах оказались так близки, возникло подозрение, что корень проблемы – недообучение или какой-то дефект в подготовке обучающих данных. Одна из рекомендуемых нами стратегий отладки предполагает визуализировать самые серьезные ошибки модели. В данном случае это означало визуализацию неправильно транскрибированных примеров из обучающего набора, которым модель приписала самую высокую степень уверенности. Оказалось, что большинство таких

примеров было слишком сильно обрезано, так что из кадра выпало несколько цифр адреса. Например, после обрезки на фотографии адреса «1849» могли остаться только цифры «849». Проблему можно было бы решить, потратив несколько недель, чтобы повысить верность системы обнаружения номера дома и выделения интересующей области. Вместо этого было принято гораздо более практичное решение – просто увеличить ширину кадра, делая ее всегда больше той, что предсказывала система обнаружения номера дома. Одно лишь это изменение повысило покрытие на 10%.

И еще несколько процентов удалось выжать путем настройки гиперпараметров. Свелась она в основном к увеличению размера модели с сохранением некоторых ограничений на вычислительную стоимость. Поскольку ошибки обучения и тестирования оставались примерно равными, всегда было понимание, что недотягивание до целевых показателей качества связано с недообучением и, быть может, несколькими оставшимися проблемами в самом наборе данных.

В общем и целом проект транскрипции оказался весьма успешным, он позволил транскрибировать сотни миллионов адресов быстрее и дешевле, чем было бы возможно в случае привлечения людей.

Мы надеемся, что благодаря принципам проектирования, описанным в этой главе, количество подобных успешных начинаний будет множиться.

Глава 12

Приложения

В этой главе мы опишем, как применить глубокое обучение к компьютерному зрению, распознаванию речи, обработке естественных языков и другим задачам, представляющим коммерческий интерес. Начнем с обсуждения реализации крупномасштабных нейронных сетей, необходимых в большинстве серьезных приложений ИИ. Затем дадим обзор нескольких конкретных приложений, в которых глубокое обучение помогло найти решение. Хотя одной из основных целей глубокого обучения является проектирование алгоритмов, способных решать широкий круг задач, в настоящий момент некоторая специализация все же необходима. Например, в задачах компьютерного зрения требуется обрабатывать много входных признаков (пикселей) для каждого примера. А в задачах обработки языка нужно моделировать много возможных значений (слов из словаря) на каждый входной признак.

12.1. Крупномасштабное глубокое обучение

Глубокое обучение основано на философии коннекционизма: отдельный биологический нейрон или отдельный признак в модели машинного обучения не обладает интеллектом, но большая популяция таких нейронов или признаков, действующих совместно, может демонстрировать разумное поведение. Важно подчеркнуть, что число нейронов должно быть большим. Один из ключевых факторов, способствовавших повышению верности нейронных сетей и сложности решаемых ими задач в период начиная с 1980-х годов и по сегодняшний день, – кардинальное увеличение размера сетей. В разделе 1.2.3 мы видели, что на протяжении последних тридцати лет размер сетей рос экспоненциально, и тем не менее искусственные нейронные сети всего лишь сравнялись по размеру с нервными системами насекомых.

Поскольку размер нейронной сети играет решающую роль, для глубокого обучения необходимы высокопроизводительное оборудование и соответствующая программная инфраструктура.

12.1.1. Реализации на быстрых CPU

Традиционно обучение нейронных сетей производилось на одной машине с одним центральным процессором (CPU). Сегодня это считается недостаточным. Мы используем в основном вычисления на графических процессорах (GPU) или на CPU многих машин, связанных в единую сеть. Но прежде чем перейти на такие дорогостоящие конфигурации, исследователям пришлось немало потрудиться, чтобы доказать, что CPU уже не справляются с вычислительной нагрузкой, необходимой нейронным сетям.

Описание эффективной реализации вычислений на CPU выходит за рамки этой книги, но отметим, что тщательно продуманная реализация на CPU нескольких семейств может принести весомые плоды. Например, в 2011 году на лучших в то время CPU код нейронных сетей выполнялся быстрее при использовании арифметики с фиксированной, а не с плавающей точкой. В работе Vanhoucke et al. (2011) благодаря тщательно оптимизированной реализации с фиксированной точкой удалось добиться трехкратного ускорения по сравнению с системой на базе арифметики с плавающей точкой. Характеристики каждой новой модели CPU отличаются от предыдущей, поэтому иногда можно ускорить и реализации с плавающей точкой. Принцип остается общим – специализация численных процедур может дать значительный выигрыш. Есть и другие стратегии, в т. ч. оптимизация структур данных с целью избежать промахов кэша и использование векторных команд. Многие исследователи, занимающиеся машинным обучением, пренебрегают такими деталями реализации, но если производительность реализации ограничивает размер модели, то страдает верность модели.

12.1.2. Реализации на GPU

Большинство современных реализаций нейронных сетей основано на использовании графических процессоров (GPU) – специализированного оборудования, которое изначально разрабатывалось для графических приложений. Потребительский рынок систем для видеоигр подстегнул создание графических карт. Как выяснилось, характеристики производительности, необходимые для игровых систем, подходят и для нейронных сетей.

Для отрисовки экрана в видеоиграх необходимо быстро выполнять много параллельных операций. Модели персонажей и окружения описываются в виде списков трехмерных координат вершин. Графическая карта должна параллельно выполнять умножение и деление матриц в большом числе вершин для преобразования трехмерных координат сцены в двумерные экранные координаты. Затем карта должна параллельно выполнить вычисления в каждом пикселе, чтобы определить его цвет. В обоих случаях вычисления довольно простые и не содержат такого большого количества ветвлений, как в типичной программе, выполняемой на CPU. Например, каждая вершина одного твердого тела умножается на одну и ту же матрицу; не нужно выполнять в каждой вершине предложение if, чтобы понять, на какую матрицу умножать. К тому же вычисления абсолютно независимы и, следовательно, легко распараллеливаются. Кроме того, в процессе вычислений производятся обращения к большим буферам в памяти, в которых хранятся растры, описывающие текстуру (цветовой узор) каждого отрисовываемого объекта. Поэтому при проектировании графических карт закладываются высокая степень параллелизма и пропускная способность памяти, а расплачиваться за это приходится меньшей тактовой частотой и не столь развитыми средствами ветвления, как в традиционных процессорах.

Алгоритмам нейронных сетей свойственны такие же характеристики производительности, как для графических алгоритмов реального времени. В нейронных сетях обычно имеется много больших буферов параметров, значений активации и градиентов, каждый из которых необходимо полностью обновлять на каждом шаге обучения. Эти буферы не помещаются в кэш стандартного настольного компьютера, поэтому пропускная способность памяти часто оказывается лимитирующим фактором. Важное преимущество GPU над CPU заключается именно в высокой пропускной способно-

сти памяти. В алгоритмах обучения нейронных сетей обычно не так много ветвлений и сложных средств управления потоком вычислений, поэтому они пригодны для работы на GPU. Поскольку нейронную сеть можно разложить на множество отдельных «нейронов», обрабатываемых независимо от других нейронов в том же слое, то для их обучения средства распараллеливания, имеющиеся в GPU, оказываются весьма кстати.

Оборудование GPU первоначально было специализировано только под графические задачи. Со временем оно стало более гибким, позволяющим использовать пользовательские подпрограммы для преобразования координат вершин и назначения цветов пикселям. В принципе, не требуется, чтобы пиксели были как-то связаны с задачей отрисовки. Такие GPU можно использовать для научных расчетов, если записывать результаты вычислений в буфер значений пикселей. В работе Steinkrau et al. (2005) реализована двухслойная полносвязная нейронная сеть на GPU и отмечено трехкратное ускорение, по сравнению с эталоном, работающим на CPU. Вскоре после этого в работе Chellapilla et al. (2006) было продемонстрировано, что такую же технику можно применить для ускорения сверточных сетей с учителем.

Использование графических карт для обучения нейронных сетей испытало настоящий взрыв популярности с появлением **GPU общего назначения** (GP-GPU), которые могут исполнять произвольный код, а не только подпрограммы отрисовки. Похожий на C язык программирования CUDA, разработанный компанией NVIDIA, дал средства для написания произвольного кода. Благодаря сравнительно удобной модели программирования, массовому параллелизму и высокой пропускной способности памяти GP-GPU стали идеальной платформой для программирования нейронных сетей. Эта платформа была с восторгом принята специалистами по глубокому обучения сразу после появления (Raina et al., 2009; Ciresan et al., 2010).

Написание эффективного кода для GP-GPU остается трудной задачей, которую лучше оставить специалистам. Приемы достижения высокой производительности на GPU сильно отличаются от того, к чему мы привыкли на CPU. Например, хорошая программа для CPU обычно проектируется так, чтобы по возможности читать данные из кэша. На GPU большая часть допускающей запись памяти не кэшируется, поэтому быстрее вычислить одно и то же значение дважды, чем вычислить однократно, а потом прочитать из памяти. Код для GPU принципиально многопоточный, и потоки необходимо тщательно синхронизировать. Например, операции с памятью выполняются быстрее, если их можно **объединить**. Объединенная операция чтения или записи имеет место, когда несколько потоков могут одновременно прочитать или записать нужные им значения в составе одной транзакции доступа к памяти. Разные модели GPU умеют объединять разные последовательности операций чтения и записи. Обычно операции с памятью проще объединить, если i-й из n потоков обращается к $(i + j)$-му байту памяти и j кратно степени двойки. Точные спецификации зависят от модели GPU. Еще один важный аспект программы для GPU – следить за тем, чтобы все потоки в группе выполняли одну и ту же команду одновременно. Это означает, что реализовать ветвление на GPU трудно. Потоки разбиваются на небольшие группы, называемые **канатами** (warp). Каждый поток каната в каждом цикле выполняет одну и ту же команду, поэтому если два потока из одного каната должны пройти по разным путям в коде, то проходить их придется последовательно, а не параллельно.

В связи с трудностями написания высокопроизводительного кода для GPU работу следует построить так, чтобы избежать разработки нового GPU-кода для тестирования новых моделей или алгоритмов. Обычно для этого создают библиотеку функций

для таких операций, как свертка и умножение матриц, а затем специфицируют модели в терминах обращений к библиотеке. Например, в библиотеке машинного обучения Pylearn2 (Goodfellow et al., 2013c) все алгоритмы специфицированы в терминах обращений к библиотекам Theano (Bergstra et al., 2010; Bastien et al., 2012) и cuda-convnet (Krizhevsky, 2010), предоставляющих высокопроизводительные операции. Такое распределение обязанностей заодно упрощает поддержку разнородного оборудования. Например, одна и та же программа, использующая Theano, может работать как на CPU, так и на GPU безо всякого изменения обращений к самой Theano. Другие библиотеки, например TensorFlow (Abadi et al., 2015) и Torch (Collobert et al., 2011b), предлагают похожие возможности.

12.1.3. Крупномасштабные распределенные реализации

Часто вычислительных ресурсов одной машины недостаточно. Поэтому хотелось бы распределить рабочую нагрузку обучения и вывода между несколькими машинами.

Распределить вывод просто, потому что каждый входной пример можно обработать на отдельной машине. Это называется **распараллеливанием по данным**.

Существует также режим **распараллеливания модели**, когда несколько машин совместно работает над одним примером, причем каждая машина выполняет свою часть модели. Это возможно как для обучения, так и для вывода.

Распараллеливание по данным на этапе обучения несколько сложнее. Мы можем увеличить размер мини-пакета, используемого на одном шаге СГС, но обычно выигрыш в терминах производительности оптимизации получается меньше линейного. Было бы лучше, чтобы несколько машин параллельно вычисляли несколько шагов градиентного спуска. К сожалению, в стандартной формулировке алгоритм градиентного спуска строго последовательный: градиент на шаге t является функцией параметров, найденных на шаге $t - 1$.

Эту проблему можно решить с помощью асинхронного стохастического градиентного спуска (Bengio et al., 2001; Recht et al., 2011). В этом случае несколько процессорных ядер сообща использует память для представления параметров. Каждое ядро читает параметры без блокировки, затем вычисляет градиент, после чего инкрементирует параметры без блокировки. Это уменьшает среднее улучшение на каждом шаге градиентного спуска, потому что некоторые ядра перезаписывают достигнутое другими ядрами, но за счет увеличенного темпа выполнения шагов процесс обучения в целом протекает быстрее. В работе Dean et al. (2012) этот подход к градиентному спуску без блокировок впервые реализован на нескольких машинах, когда параметры хранятся не в разделяемой памяти, а на сервере параметров. Распределенный асинхронный градиентный спуск остается основной стратегией обучения больших глубоких сетей и применяется большинством групп, занимающих лидирующие позиции в индустрии (Chilimbi et al., 2014; Wu et al., 2015). Академические учреждения обычно не могут позволить себе распределенные системы обучения такого масштаба, но в ряде исследований изучается вопрос о построении распределенных сетей на сравнительно дешевом оборудовании, имеющемся в университетах (Coates et al., 2013).

12.1.4. Сжатие модели

Во многих коммерческих приложениях гораздо важнее, чтобы время и потребление памяти были низкими на этапе вывода модели машинного обучения, тогда как затраты на этапе обучения не столь критичны. Если приложение не требует персо-

нализации, то модель можно обучить один раз, а затем развернуть для миллиардов пользователей. Во многих случаях у конечного пользователя ресурсов меньше, чем у разработчика. Например, сеть распознавания речи можно обучить на кластере из мощных компьютеров, а затем установить в мобильный телефон.

Ключевая стратегия сокращения стоимости вывода – **сжатие модели** (Buciluǎ et al., 2006). Основная идея – заменить исходную дорогостоящую модель другой, потребляющей меньше памяти и работающей быстрее.

Сжатие модели применимо, когда размер исходной модели обусловлен в основном стремлением предотвратить переобучение. В большинстве случаев модель с наименьшей ошибкой обобщения представляет собой ансамбль нескольких независимо обученных моделей. Вычислять предсказание всех n членов ансамбля дорого. Иногда даже одна модель обобщается лучше, если она велика (например, если применялась регуляризация прореживанием).

Такие большие модели обучают некоторую функцию $f(\boldsymbol{x})$, но используют при этом гораздо больше параметров, чем необходимо для решения задачи. Их размер велик только потому, что число обучающих примеров ограничено. После того как функция $f(\boldsymbol{x})$ аппроксимирована, мы можем сгенерировать обучающий набор, содержащий бесконечно много примеров, просто применив f к случайной выборке $\boldsymbol{x}$. Затем мы обучаем новую, меньшую модель, так чтобы она совпадала с $f(\boldsymbol{x})$ на этой выборке. Чтобы емкость новой модели использовалась наиболее эффективно, лучше выбирать новые точки $\boldsymbol{x}$ из распределения, похожего на реальные тестовые данные, которые будут предъявлены модели впоследствии. Это можно сделать, слегка исказив обучающие примеры или произведя выборку из порождающей модели, обученной на исходном обучающем наборе.

Альтернативно можно обучить меньшую модель только на исходных обучающих примерах, но научить ее копировать другие признаки модели, например апостериорное распределение неправильных классов (Hinton et al., 2014, 2015).

12.1.5. Динамическая структура

Одна из общих стратегий ускорения систем обработки данных – построить систему с **динамической структурой** в виде графа, описывающего вычисления, необходимые для обработки входных данных. Системы обработки данных могут динамически определить, какую часть большого множества нейронных сетей выполнять для заданного входа. Отдельные нейронные сети также могут иметь внутреннюю динамическую структуру, т. е. определять, какое подмножество признаков (скрытых блоков) вычислять при имеющейся входной информации. Такую форму динамической структуры внутри нейронной сети иногда называют **условным вычислением** (Bengio, 2013; Bengio et al., 2013b). Поскольку многие компоненты архитектуры могут иметь отношение только к небольшому количеству возможных входов, система будет работать быстрее, если эти признаки вычисляются только по мере необходимости.

Динамическая структура вычислений – базовый принцип информатики, повсеместно применяемый в программной инженерии. Простейшие варианты динамической структуры в контексте нейронных сетей основаны на определении того, какое подмножество некоторой группы нейронных сетей (или иных моделей машинного обучения) следует применить к конкретному входу.

Давно сложившаяся стратегия ускорения вывода – использовать **каскад** классификаторов. Ее можно применить, когда стоит задача обнаружить присутствие ред-

кого объекта (или события). Для полной уверенности мы должны использовать изощренный классификатор высокой емкости, а его работа обходится дорого. Но поскольку объект редкий, мы зачастую можем ограничиться куда меньшими вычислениями, чтобы отклонить входы, не содержащие объекты. В такой ситуации можно обучить последовательность классификаторов. У начальных классификаторов емкость низкая, а при их обучении предпочтение отдается полноте. Иными словами, они не отклоняют по ошибке вход, если объект присутствует. При обучении последнего классификатора на первое место ставится точность. На этапе тестирования мы последовательно выполняем классификаторы и отклоняем пример, как только его отвергнет первый же классификатор. Вообще говоря, это позволяет с высокой уверенностью проверить присутствие объекта, пользуясь моделью высокой емкости, но при этом не платить за полный вывод для каждого примера. Есть два способа обеспечить высокую емкость каскада. Первый – сделать так, чтобы все члены, находящиеся ближе к концу каскада, имели высокую емкость. Очевидно, что тогда и система в целом будет иметь высокую емкость, поскольку таковы некоторые ее компоненты. Можно поступить иначе – построить каскад, в котором емкость каждой отдельной модели низкая, но система в целом обладает высокой емкостью, будучи комбинацией большого числа малых моделей. В работе Viola and Jones (2001) каскад усиленных решающих деревьев использовался для построения быстрого и устойчивого детектора лиц, пригодного для использования в портативных цифровых камерах. В созданном ими классификаторе для локализации лица применяется метод скользящего окна: исследуется много прямоугольных окон и отбрасываются те, в которых лиц нет. Еще в одной версии каскада предшествующие модели используются для реализации своего рода механизма внимания: первые члены каскада локализуют объект, а последующие выполняют дополнительную обработку, зная, где объект расположен. Например, Google транскрибирует номера домов на изображениях Street View с помощью двухступенчатого каскада, в котором для нахождения номера дома применяется одна модель машинного обучения, а для транскрипции – другая (Goodfellow et al., 2014d).

Решающие деревья сами по себе являются примером динамической структуры, поскольку каждый узел дерева определяет, какое поддерево вычислять для поступившего входа. Чтобы объединить глубокое обучение с динамической структурой, можно, например, обучить решающее дерево, в каждом узле которого имеется нейронная сеть, принимающая решение о разделении (Guo and Gelfand, 1992), хотя обычно главная цель такого подхода не в том, чтобы ускорить вывод.

Продолжая в том же духе, можно использовать нейронную сеть, называемую **привратником** (gater), для выбора одной из нескольких **экспертных сетей**, вычисляющих выход по текущему входу. Первый вариант этой идеи получил название «**коллектив экспертов**» (mixture of experts) (Nowlan, 1990; Jacobs et al., 1991) – привратник выводит набор вероятностей или весов (полученных с помощью нелинейности типа softmax), по одному на каждого эксперта, а конечный выход получается взвешиванием выходов экспертов. В этом случае применение привратника не снижает вычислительную стоимость, но если бы для каждого примера привратник предлагал только одного эксперта, то мы получили бы **строгий** (hard) **коллектив экспертов** (Collobert et al., 2001, 2002), способный значительно ускорить обучение и вывод. Такая стратегия хорошо работает, когда число решений привратника мало, а не растет комбинаторно. Но если мы хотим выбирать различные подмножества блоков или параметров, то воспользоваться «мягким переключателем» не получится, потому что требуется

перечислить все конфигурации привратника (и вычислить для них выходы). Чтобы справиться с этой проблемой, было предложено несколько подходов к обучению комбинаторных привратников. В работе Bengio et al. (2013b) описаны эксперименты с несколькими оценками градиента по вероятностям привратника, а в работах Bacon et al. (2015) и Bengio et al. (2015a) использовалась техника обучения с подкреплением (градиент политики – policy gradient) для реализации разновидности условного прореживания, применяемого к группам скрытых блоков, с целью реально снизить вычислительную стоимость, не ухудшив качества аппроксимации.

Еще один вид динамической структуры – переключатель, когда скрытый блок может получать входные данные от разных блоков в зависимости от контекста. Такую динамическую маршрутизацию можно интерпретировать как механизм внимания (Olshausen et al., 1993). Пока что эффективность жесткого переключателя в крупномасштабных приложениях не доказана. Вместо этого в современных системах применяется взвешенное среднее многих возможных входов, что не позволяет реализовать все потенциальные вычислительные преимущества динамической структуры. Современные механизмы внимания описаны в разделе 12.4.5.1.

Одно из главных препятствий на пути использования систем с динамической структурой – снижение степени параллелизма из-за того, что выполнение идет по разным ветвям для разных входов. Это означает, что лишь немногие операции сети можно описать как умножение матриц или пакетную свертку на мини-пакете примеров. Можно написать более специализированные подпрограммы, которые сворачивают каждый пример с разными ядрами или умножают каждую строку матриц плана на разное подмножество столбцов матрицы весов. К сожалению, такие подпрограммы трудно реализовать эффективно. Реализации для CPU будут медленными из-за отсутствия когерентности кэшей, а реализации для GPU – тоже медленными из-за отсутствия объединенных транзакций доступа к памяти и необходимости сериализовывать канаты, когда входящие в них потоки выполняют разные ветви программы. Иногда эти проблемы можно смягчить, разбив примеры на группы, для которых выполнение идет по одной ветви, а затем обработав каждую группу одновременно. Такая стратегия может оказаться приемлемой для минимизации времени обработки фиксированного числа примеров в офлайновой конфигурации. Но в режиме реального времени, когда примеры должны обрабатываться непрерывно, разбиение задачи на части может привести к проблемам с балансированием нагрузки. Например, если назначить одну машину для обработки первой ступени каскада, а другую – для обработки последней, то первая машина может оказаться перегруженной, а вторая – недогруженной. Аналогичные проблемы возникают, если разным машинам поручить реализацию различных узлов нейронного решающего дерева.

12.1.6. Специализированные аппаратные реализации глубоких сетей

Когда исследования по нейронным сетям только начинались, инженеры-электроники уже задумались над специализированным оборудованием, способным ускорить обучение или вывод. Ранние и сравнительно недавние обзоры специализированного оборудования для глубоких сетей можно найти в работах Lindsey and Lindblad, 1994, Beiu et al., 2003, Misra and Saha, 2010.

За несколько десятков лет были разработаны разнообразные виды специализированного оборудования (Graf and Jackel, 1989; Mead and Ismail, 2012; Kim et al., 2009;

Pham et al., 2012; Chen et al., 2014a,b) на базе ASIC (интегральных схем специального назначения): цифровых, аналоговых (Graf and Jackel, 1989; Mead and Ismail, 2012) или гибридных. В последние годы фокус сместился в сторону более гибких реализаций на основе ППВМ (программируемых пользователем вентильных матриц), допускающих конфигурацию пользователем после изготовления.

Хотя в программных реализациях для процессоров общего назначения (CPU и GPU) обычно используются 32- или 64-разрядные числа с плавающей точкой, давно известно, что можно обойтись и меньшей точностью, по крайней мере на этапе вывода (Holt and Baker, 1991; Holi and Hwang, 1993; Presley and Haggard, 1994; Simard and Graf, 1994; Wawrzynek et al., 1996; Savich et al., 2007). В последние годы этот вопрос приобрел особую остроту, поскольку глубокое обучение стало широко использоваться в промышленных изделиях, а на примере GPU продемонстрирован серьезный выигрыш, который может дать более быстрое оборудование. Еще одним фактором, стимулирующим исследования в области специализированного оборудования для глубокого обучения, является замедление прогресса в разработке одного ядра CPU и GPU; теперь повышение быстродействия происходит в основном за счет распараллеливания обработки между несколькими ядрами (как в CPU, так и в GPU). Эта ситуация сильно отличается от сложившейся в 1990-е годы (время предыдущего поколения нейронных сетей), когда аппаратные реализации нейронных сетей (на создание которых могло уйти до двух лет с начала проекта до выпуска готовой микросхемы) не могли конкурировать с CPU общего назначения ни по темпам развития, ни по стоимости. Таким образом, специализированное оборудование – это способ выйти за привычные рамки во времена, когда проектируется новое оборудование для устройств с низким энергопотреблением (например, смартфонов), а его цель – сделать общедоступными приложения глубокого обучения (распознавание речи, компьютерное зрение, обработка естественных языков).

Недавние работы по реализации нейронных сетей с обратным распространением на оборудовании с арифметикой низкой точности (Vanhoucke et al., 2011; Courbariaux et al., 2015; Gupta et al., 2015) позволяют сделать вывод, что для обучения и использования таких сетей достаточно от 8 до 16 разрядов. Ясно также, что на этапе обучения нужна большая точность, чем на этапе вывода, и что для снижения разрядности можно использовать некоторые формы динамического представления чисел с фиксированной точкой. Традиционно числа с фиксированной точкой занимают фиксированный диапазон (как если бы зафиксировать показатель степени в представлении с плавающей точкой). Динамические представления с фиксированной точкой позволяют разделить этот диапазон между множеством чисел (например, весов в одном слое). Использование фиксированной точки вместо плавающей и понижение разрядности уменьшают площадь, занимаемую оборудованием, энергопотребление и время выполнения операции умножения, а именно на эти операции приходится основное время при обучении и использовании современной глубокой сети с обратным распространением.

12.2. Компьютерное зрение

Компьютерное зрение традиционно является одной из основных областей применения глубокого обучения, потому что зрение не требует никаких усилий от человека и многих животных, но является сложной задачей для компьютеров (Ballard et al.,

1983). Многие из самых популярных стандартных эталонных задач для сравнения алгоритмов глубокого обучения так или иначе связаны с распознаванием объектов или текста.

Компьютерное зрение – это очень широкая дисциплина, охватывающая различные способы обработки изображений и поразительное разнообразие приложений – от воспроизведения зрительных способностей человека, например распознавания лиц, до создания совершенно новых видов зрительных способностей. В качестве примера из последней категории можно назвать недавнее приложение по распознаванию звуковых волн по вибрациям, которые они вызывают в предметах, присутствующих на видео (Davis et al., 2014). Большая часть исследований по глубокому обучению в контексте компьютерного зрения посвящена не таким экзотическим приложениям, расширяющим горизонты возможного, а сравнительно узким базовым задачам ИИ – воспроизведению человеческих способностей. Речь идет о той или иной форме обнаружения или распознавания объектов: констатация присутствия объекта в изображении, обведение найденных в изображении объектов рамочкой, транскрибирование последовательности символов в изображении или пометка каждого пикселя изображения идентификатором объекта, которому он принадлежит. Поскольку основополагающим принципом глубокого обучения изначально являлось порождающее моделирование, существует много работ по синтезу изображений с помощью глубоких моделей. Синтез изображений из ничего обычно не считается крупным достижением в компьютерном зрении, но модели, способные к синтезу, часто бывают полезными для восстановления изображений – задачи компьютерного зрения, подразумевающей исправление дефектов и удаление объектов из изображения.

12.2.1. Предобработка

Во многих приложениях требуется изощренная предварительная обработка, потому что исходные данные представлены в виде, малопригодном для архитектур глубокого обучения. В компьютерном зрении объем такой предобработки сравнительно невелик. Изображения следует привести к стандартному виду, так чтобы значения всех пикселей принадлежали одному и тому же разумному диапазону, например [0,1] или [–1, 1]. Если в одних изображениях пиксели лежат в диапазоне [0,1], а в других – в диапазоне [0, 255], то попытка их обработать обычно не приводит ни к чему хорошему. Во многих архитектурах компьютерного зрения требуется, чтобы все изображения были стандартного размера, поэтому их необходимо либо обрезать, либо масштабировать. Но даже такое масштабирование не всегда необходимо. Некоторые сверточные модели принимают входные данные переменного размера и динамически подстраивают свои области пулинга, так чтобы размер выхода был постоянным (Waibel et al., 1989). В других сверточных моделях размер выхода переменный, автоматически подстраиваемый под размер входа, например в моделях, предназначенных для очистки от шумов или для пометки каждого пикселя изображения (Hadsell et al., 2007).

Пополнение набора данных можно рассматривать как вид предобработки одного лишь обучающего набора. В большинстве моделей машинного зрения это отличный способ уменьшить ошибку обобщения. На этапе тестирования применима похожая идея: предъявить много разных вариантов одного и того же входа (например, одно изображение, обрезанное немного по-разному) и организовать голосование разных

экземпляров модели для определения выхода. Это можно интерпретировать как разновидность ансамблевого подхода, уменьшающего ошибку обобщения.

Другие виды предобработки применяются как к обучающему, так и к тестовому набору с целью привести каждый пример к канонической форме, уменьшив тем самым вариативность, которую модель должна учитывать. Снижение степени вариативности способствует уменьшению ошибки обобщения и размера модели, необходимого для аппроксимации обучающего набора. Для решения более простых задач достаточно моделей меньшего размера, а чем проще решение, тем больше шансов, что оно хорошо обобщается. Предобработку такого рода обычно проектируют так, чтобы устранить вариативность, которую проектировщик может легко описать и которая точно не имеет существенной связи с основной задачей. При обучении больших моделей на больших наборах данных такая предобработка часто излишня, и лучше дать модели возможность обучиться всем видам вариативности, к которым она должна быть инвариантна. Например, в системе AlexNet для классификации набора ImageNet есть только один шаг предобработки: вычитание среднего значения каждого пикселя, вычисленного по всем обучающим примерам (Krizhevsky et al., 2012).

12.2.1.1. Нормализация контрастности

Один из самых очевидных источников вариативности, который во многих задачах можно без опаски устранить, – контрастность изображения, т. е. абсолютная величина разности между яркими и темными пикселями. Существует много способов выразить контрастность количественно. В контексте глубокого обучения обычно контрастностью называют стандартное отклонение пикселей всего изображения или его части. Допустим, что изображение представлено тензором $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$, где $X_{i,j,1}$ – яркость красного в точке на пересечении i-й строки и j-го столбца, $X_{i,j,2}$ – яркость зеленого, а $X_{i,j,3}$ – яркость синего. Тогда контрастность всего изображения равна

$$\sqrt{\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k}-\bar{\mathbf{X}})^2}, \tag{12.1}$$

где $\bar{\mathbf{X}}$ – средняя яркость всего изображения:

$$\bar{\mathbf{X}} = \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}X_{i,j,k}. \tag{12.2}$$

Цель **глобальной нормализации контрастности** (ГНК) – уравнять контрастность изображения; для этого из каждого пикселя вычитается среднее значение по всему изображению, а затем производится нормировка изображения, так чтобы стандартное отклонение по всем пикселям было равно некоторой константе s. Дополнительная сложность заключается в том, что ни при каком нормировочном коэффициенте невозможно изменить контрастность изображения с нулевой контрастностью (когда яркость всех пикселей одинакова). Изображения с очень низкой, но ненулевой контрастностью обычно не несут почти никакого информационного содержания. Деление на истинное стандартное отклонение ничего не дает, а только усиливает шум сенсоров или артефакты сжатия. Поэтому вводится небольшой положительный регуляризационный параметр λ, чтобы сместить оценку стандартного отклонения. Вместо этого можно наложить ограничение на знаменатель: не менее ε. Если дано входное изображение $\mathbf{X}$, то в результате ГНК получается выходное изображение $\mathbf{X}'$, определенное следующим образом:

$$X'_{i,j,k} = s\frac{X_{i,j,k} - \bar{X}}{\max\left\{\varepsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k} - \bar{X})^2}\right\}}. \tag{12.3}$$

В наборе данных, состоящем из больших изображений, обрезанных по границе интересующих объектов, вряд ли имеются изображения с почти постоянной яркостью. В таком случае можно без опаски игнорировать проблему малого знаменателя, положив $\lambda = 0$, и избежать деления на 0, возможного в редчайших случаях, взяв в качестве ε какое-нибудь очень малое значение, например 10^{-8}. Именно такой подход принят в работе Goodfellow et al. (2013a) для набора данных CIFAR-10. Для небольших случайно обрезанных изображений шансы получить почти постоянную яркость выше, поэтому агрессивная регуляризация имеет больше смысла. В работе Coates et al. (2011) взяты значения $\varepsilon = 0$, $\lambda = 10$ для небольших случайно выбранных фрагментов изображений из набора CIFAR-10.

Параметр масштабирования s обычно можно положить равным 1, как сделано в работе Coates et al. (2011), или выбрать так, чтобы стандартное отклонение каждого отдельного пикселя по всем примерам было близко к 1, как в работе Goodfellow et al. (2013a).

Стандартное отклонение в формуле (12.3) – это просто умноженная на коэффициент норма L^2 изображения (в предположении, что среднее уже вычтено). ГНК предпочтительнее определять в терминах стандартного отклонения, а не нормы L^2, поскольку первое включает деление на число пикселей, так что одно и то же значение s можно использовать независимо от размера изображения. Однако тот факт, что норма L^2 пропорциональна стандартному отклонению, позволяет сделать некоторые интуитивные заключения. ГНК можно интерпретировать как отображение примеров на сферическую оболочку, как показано на рис. 12.1. Это полезное свойство, потому что нейронные сети часто лучше откликаются на пространственные направления, а не на точные позиции. Для отклика на различные расстояния в одном направлении необходимы скрытые блоки с коллинеарными векторами весов, но разными смещениями. Такую взаимосвязь алгоритму обучения выявить трудно. Кроме того, у многих мелких графических моделей возникают трудности, когда нужно представить несколько раздельных мод на одной прямой. ГНК предотвращает такие проблемы, сводя каждый пример к направлению, а не к комбинации направления и расстояния.

В противоречие с интуицией существует операция предобработки, называемая **сферингом** (sphering), отличающаяся от ГНК. Результатом сферинга является не отображение данных на поверхность сферы, а нормировка главных компонент на одинаковую дисперсию, так чтобы многомерное нормальное распределение, используемое в алгоритме PCA, имело сферические контуры. Операция сферинга больше известна под названием **отбеливание** (whitening).

Применение глобальной нормализации контрастности зачастую не выделяет тех признаков изображения, которые мы хотели бы подчеркнуть, например границ и углов. Если на сцене имеется большая темная и большая светлая область (например, городской сквер, половина которого находится в тени здания), то ГНК гарантирует значительную разницу между яркостью темной и светлой областей. Однако нет никакой гарантии, что границы внутри темной области будут выделяться.

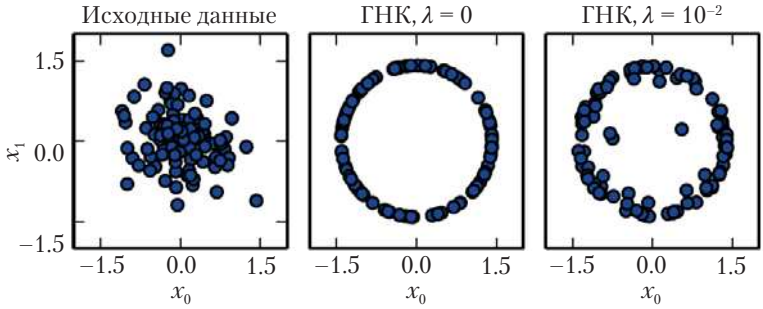

Рис. 12.1 ❖ ГНК отображает примеры на сферу. (*Слева*) У исходных данных норма может быть любой. (*В центре*) ГНК с $\lambda = 0$ отображает все ненулевые примеры на поверхность сферы. В данном случае $s = 1$ и $\varepsilon = 10^{-8}$. Поскольку ГНК основана на стандартном отклонении, а не на норме L^2, эта сфера не единичная. (*Справа*) Регуляризированная ГНК с $\lambda > 0$ сдвигает примеры в сторону сферы, но не полностью устраняет дисперсию нормы. Значения s и λ те же, что и раньше

Поэтому нам нужна также **локальная нормализация контрастности**. Она гарантирует нормализацию контрастности в каждом небольшом окне, а не во всем изображении. На рис. 12.2 приведено сравнение глобальной и локальной нормализаций контрастности.

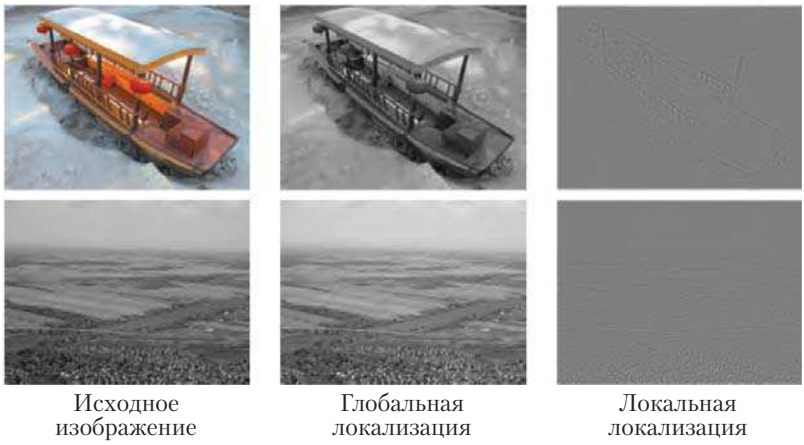

Рис. 12.2 ❖ Сравнение глобальной и локальной нормализаций контрастности. Визуально эффект глобальной нормализации почти не заметен. Все изображения приводятся примерно к одной шкале, чтобы не заставлять алгоритм обучения обрабатывать несколько шкал. Локальная нормализация контрастности изменяет изображение гораздо сильнее, отбрасывая все области постоянной яркости. В результате модель может сосредоточиться только на границах. Области с мелкой текстурой, например дома во втором ряду, могут утратить детали, если ширина ядра нормализации слишком велика

Локальную нормализацию контрастности можно определить по-разному. В любом случае модифицируется каждый пиксель: из него вычитается среднее близлежащих

пикселей, и результат делится на стандартное отклонение близлежащих пикселей. Иногда речь идет о настоящем среднем и стандартном отклонениях по прямоугольному окну с центром в модифицируемом пикселе (Pinto et al., 2008). А иногда берется взвешенное среднее и взвешенное стандартное отклонения с весами, имеющими нормальное распределение с центром в модифицируемом пикселе. Для цветных изображений есть разные стратегии: в одних цветовые каналы обрабатываются раздельно, в других информация из разных цветовых каналов комбинируется при нормализации каждого пикселя (Sermanet et al., 2012).

Обычно локальную нормализацию контрастности можно эффективно реализовать с помощью сепарабельной свертки (см. раздел 9.8), которая вычисляет карты признаков локальных средних и локальных стандартных отклонений, с последующим поэлементным вычитанием и поэлементным делением разных карт признаков.

Локальная нормализация контрастности – дифференцируемая операция, ее можно использовать как в качестве нелинейности, применяемой к скрытым слоям сети, так и в качестве операции предобработки, применяемой к входному изображению.

Как и в случае глобальной нормализации контрастности, обычно необходимо регуляризировать локальную нормализацию, чтобы избежать деления на нуль. А поскольку локальная нормализация применяется к меньшим окнам, то регуляризация даже более важна. В малом окне больше шансов, что значения всех пикселей будут примерно одинаковы, и, стало быть, больше вероятность получить нулевое стандартное отклонение.

12.2.1.2. Пополнение набора данных

Как было сказано в разделе 7.4, повысить обобщаемость классификатора можно, просто увеличив размер обучающего набора путем добавления копий примеров, которые были модифицированы с помощью преобразований, не изменяющих класса. Такой вид пополнения набора данных особенно хорош в применении к распознаванию объектов, поскольку класс инвариантен к самым разным преобразованиям, так что ко входу можно применить различные геометрические операции. Как уже отмечалось, классификатор можно улучшить, если включить в набор данных случайные параллельные переносы, повороты, а в некоторых случаях и отражения изображения относительно оси. В специализированных приложениях компьютерного зрения набор пополняют и результатами более сложных преобразований, например случайного возмущения цветов (Krizhevsky et al., 2012) и нелинейного геометрического искажения входа (LeCun et al., 1998b).

12.3. Распознавание речи

Задача распознавания речи состоит в том, чтобы отобразить акустический сигнал, содержащий текст, произнесенный на естественном языке, в последовательность слов. Обозначим $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(T)})$ последовательность векторов акустического входа (обычно получаемую разбиением сигнала на отрезки длительностью 20 мс). В большинстве систем распознавания речи для предобработки входа используются специальные спроектированные вручную признаки, но некоторые системы глубокого обучения (Jaitly and Hinton, 2011) обучаются признакам прямо на исходных данных. Обозначим $\boldsymbol{y} = (y_1, y_2, ..., y_N)$ целевую выходную последовательность (обычно последовательность слов или символов). Задача **автоматического распознавания речи**

(APP, англ. ASR) – создать функцию f^*_{ASR}, вычисляющую наиболее вероятную лингвистическую последовательность $\boldsymbol{y}$ по заданной акустической последовательности $\boldsymbol{X}$:

$$f^*_{\text{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\mathbf{y} \mid \mathbf{X} = \boldsymbol{X}), \tag{12.4}$$

где P^* – истинное условное распределение, связывающее входы $\boldsymbol{X}$ с выходами $\boldsymbol{y}$.

Начиная с 1980-х годов и примерно до 2009–2012-го системы распознавания речи строились в основном с помощью комбинации скрытых марковских моделей (СММ) и моделей гауссовых смесей (МГС). МГС моделируют ассоциацию между акустическими признаками и фонемами (Bahl et al., 1987), а СММ – последовательность фонем. В семействе моделей СММ-МГС акустические сигналы рассматриваются как порожденные следующим процессом: сначала СММ генерирует последовательность фонем и подфонемных состояний (начало, середина и конец каждой фонемы), затем МГС преобразует каждый дискретный символ в короткий сегмент акустического сигнала. Системы СММ-МГС еще недавно занимали доминирующее положение, но распознавание речи стало одной из первых областей применений нейронных сетей, и во многих системах APP, созданных в конце 1980-х и в начале 1990-х годов, они использовались (Bourlard and Wellekens, 1989; Waibel et al., 1989; Robinson and Fallside, 1991; Bengio et al., 1991, 1992; Konig et al., 1996). В то время качество систем APP на основе нейронных сетей было приблизительно таким же, как систем СММ-МГС. Например, в работе Robinson and Fallside (1991) была достигнута частота ошибок распознавания фонем 26% на корпусе текстов TIMIT (Garofolo et al., 1993), содержащем 39 различимых фонем, что было сравнимо с СММ-системами или даже лучше. С тех пор корпус TIMIT стал эталоном для распознавания фонем и играет такую же роль, как набор данных MNIST для распознавания объектов. Тем не менее из-за технических сложностей построения программных систем для распознавания речи и тех усилий, которые были вложены в создание таких систем на основе СММ-МГС, индустрия не увидела убедительных аргументов в пользу перехода на нейронные сети. Поэтому до конца 2000-х проводимые в академических и промышленных кругах исследования по применению нейронных сетей к распознаванию речи были в основном сосредоточены на обучении дополнительных признаков для систем СММ-МГС.

Позже, когда модели стали *гораздо больше и глубже*, а размер наборов данных резко увеличился, верность распознавания удалось значительно повысить, используя нейронные сети вместо МГС для ассоциированния акустических признаков с фонемами (или подфонемными состояниями). Начиная с 2009 года ученые применили к распознаванию речи вариант глубокого обучения на базе обучения без учителя. В основе этого подхода лежало применение неориентированных вероятностных моделей, называемых ограниченными машинами Больцмана (ОМБ, англ. RBM), к моделированию входных данных. ОМБ описаны в третьей части книги. Для распознавания речи использовали предобучение без учителя для построения глубокой сети прямого распространения, слои которой инициализировались посредством обучения ОМБ. Эти сети принимали представление акустического спектра во входном окне фиксированного размера (вокруг центрального кадра) и предсказывали условные вероятности состояний СММ для этого центрального кадра. Обучение глубоких сетей позволило значительно повысить частоту распознавания на корпусе TIMIT (Mohamed et al., 2009, 2012a), снизив частоту ошибок с 26 до 20.7%. В работе Mohamed et al. (2012b) проанализированы причины успеха таких моделей. Распространение на конвейер распознавания в телефоне привело к добавлению адаптивных признаков (Mohamed

et al., 2011), что позволило еще снизить частоту ошибок. Затем быстро последовали работы по обобщению архитектуры с распознавания фонем (тема корпуса TIMIT) на распознавание речи с большим словарем (Dahl et al., 2012), т. е. распознавание последовательностей слов, взятых из большого словаря. В конечном итоге акцент в применении глубоких сетей для распознавания речи сместился с предобучения и машин Больцмана на такие темы, как блоки линейной ректификации и прореживание (Zeiler et al., 2013; Dahl et al., 2013). К тому времени большинство известных коллективов в промышленности приступило к исследованию глубокого обучения совместно с учеными из академического сообщества. В работе Hinton et al. (2012a) описаны прорывы, которых удалось достичь в результате такого сотрудничества, сейчас они внедрены в смартфоны и другие изделия.

Впоследствии, по мере увеличения размеченных наборов данных и включения некоторых методов инициализации, обучения и настройки архитектуры глубоких сетей, эти группы пришли к выводу, что этап предобучения без учителя либо излишний, либо не дает существенного улучшения.

Качественный прорыв в терминах частоты ошибок распознавания слов был беспрецедентным (около 30%), и за ним последовал длительный период, примерно 10 лет, в течение которого применение традиционной технологии СММ-МГС не приводило к существенному снижению частоты ошибок, несмотря на постоянно увеличивающийся размер обучающих наборов (см. рис. 2.4 в работе Deng and Yu [2014]). В результате исследователи в области распознавания речи обратили взоры в сторону глубокого обучения. Не прошло и двух лет, как большинство коммерческих продуктов для распознавания речи включало глубокие нейронные сети, что подхлестнуло новые исследования по алгоритмам глубокого обучения и архитектурам АРР. Эти исследования продолжаются и по сей день.

Одним из новшеств стало использование сверточных сетей (Sainath et al., 2013), реплицирующих веса по времени и частоте, что привело к улучшению качества, по сравнению с нейронными сетями с временной задержкой, в которых веса реплицировались только по времени. В новых двумерных сверточных моделях входная спектрограмма рассматривалась не как один длинный вектор, а как изображение, одна ось которого соответствует времени, а другая – частоте спектральных составляющих.

Еще одно важное направление, работы в котором по-прежнему ведутся, – переход к сквозным системам распознавания речи на базе глубокого обучения, из которых вообще устранены СММ. Первым значительным прорывом в этом направлении стала работа Graves et al. (2013), в которой обучена глубокая рекуррентная нейронная сеть LSTM (см. раздел 10.10), использующая вывод на базе максимума апостериорной вероятности поверх совмещения кадров с фонемами, как в работе LeCun et al. (1998b) и в системе CTC (Graves et al., 2006; Graves, 2012). В глубокой РНС (Graves et al., 2013) имеются переменные состояния из нескольких слоев на каждом временном шаге, что придает развернутому графу два вида глубины: обычную, обусловленную наличием нескольких слоев, и вследствие развертки во времени. В этой работе частоту ошибок распознавания фонем на корпусе TIMIT удалось снизить до рекордных 17.7%. О вариантах глубоких РНС, применяемых в других ситуациях, см. работы Pascanu et al. (2014a) и Chung et al. (2014).

Еще один недавний шаг в сторону сквозного глубокого обучения систем АРР – научить систему «совмещать» акустическую информацию с фонетической (Chorowski et al., 2014; Lu et al., 2015).

12.4. Обработка естественных языков

Под **обработкой естественных языков** (ОЕЯ, англ. NLP) понимается использование таких языков, как английский или русский, компьютером. Компьютерные программы обычно читают и порождают тексты на искусственных языках, спроектированных с целью обеспечить эффективный и однозначный грамматический разбор. Естественные языки зачастую неоднозначны и не поддаются формальному описанию. К сфере обработки естественных языков относятся такие приложения, как машинный перевод, когда обучаемая система читает предложение на одном языке и порождает эквивалентное ему на другом языке. Многие приложения ОЕЯ основаны на языковых моделях, в которых определено распределение вероятности последовательностей слов, символов и байтов в естественном языке.

Как и в случае других обсуждаемых в этой главе приложений, весьма общие нейросетевые методы можно с успехом применить и к обработке естественных языков. Но для достижения высокого качества и масштабируемости важны предметно-ориентированные стратегии. Для построения эффективной модели естественного языка обычно используются методы, специализированные для обработки последовательных данных. Во многих случаях мы предпочитаем рассматривать естественный язык как последовательность слов, а не отдельных символов или байтов. Поскольку число слов велико, словесные модели языка должны работать в разреженных дискретных пространствах очень высокой размерности. Разработано несколько стратегий обеспечения вычислительной и статистической эффективности таких моделей.

12.4.1. *n*-граммы

В **языковой модели** определено распределение вероятности последовательностей лексем естественного языка. В зависимости от вида модели лексемой может быть слово, символ или даже байт. Лексемы всегда дискретны. В самых ранних успешных языковых моделях использовались последовательности лексем фиксированной длины, называемые *n*-граммами.

В моделях на основе *n*-грамм определена условная вероятность *n*-ой лексемы при условии предыдущих $n - 1$ лексем. Произведения этих условных вероятностей определяют распределение вероятности более длинных последовательностей:

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}). \tag{12.5}$$

Это разложение – не что иное, как цепное правило вероятностей. Распределение вероятности начальной последовательности $P(x_1, \ldots, x_{n-1})$ можно смоделировать с помощью другой модели с меньшим значением n.

Обучение *n*-граммных моделей не вызывает трудностей, потому что оценку максимального правдоподобия можно вычислить, просто подсчитав, сколько раз каждая возможная *n*-грамма встречается в обучающем наборе. Модели на основе *n*-грамм были основным компонентом статистического моделирования языков в течение многих десятилетий (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999).

Для небольших значений n у *n*-грамм даже есть специальные названия: **униграмма** для $n = 1$, **биграмма** для $n = 2$ и **триграмма** для $n = 3$. Эти названия образуются из латинского префикса числительного и греческого суффикса «грамма», обозначающего нечто написанное.

Обычно модели n-грамм и $(n-1)$-грамм обучаются одновременно. Это упрощает вычисление

$$P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \ldots, x_t)}{P_{n-1}(x_{t-n+1}, \ldots, x_{t-1})} \tag{12.6}$$

– нужно лишь найти две сохраненные вероятности. Чтобы точно воспроизвести вывод в модели P_n, мы должны опустить последний символ каждой последовательности при обучении P_{n-1}.

В качестве примера продемонстрируем, как триграммная модель вычисляет вероятность предложения «THE DOG RAN AWAY» (собака убежала). Первые слова предложения нельзя обработать с помощью формулы по умолчанию, основанной на условной вероятности, потому что в начале предложения еще нет никакого контекста. Поэтому вначале используются безусловные вероятности слов. Таким образом, мы вычисляем P_3(THE DOG RAN). Последнее же слово можно предсказать стандартно, воспользовавшись условным распределением P(AWAY | DOG RAN). Подстановка в формулу (12.6) дает:

$$P(\text{THE DOG RAN AWAY}) = P_3(\text{THE DOG RAN})P_3(\text{DOG RAN AWAY})/P_2(\text{DOG RAN}). \tag{12.7}$$

Фундаментальное ограничение максимального правдоподобия в n-граммных моделях состоит в том, что оценка P_n по счетчикам в обучающем наборе во многих случаях близка к нулю, несмотря даже на то, что кортеж $(x_{t-n+1}, \ldots, x_t)$ может встречаться в тестовом наборе. Это может привести к катастрофическим последствиям двух видов. Если P_{n-1} равно нулю, то отношение не определено, поэтому модель вообще не дает разумного ответа. Если же P_{n-1} не равно нулю, но P_n равно нулю, то логарифмическая вероятность равна $-\infty$. Чтобы избежать таких неприятностей, в большинстве n-граммных моделей используется та или иная форма **сглаживания**. Смысл этого приема состоит в том, чтобы сдвинуть массу вероятности от наблюдавшихся кортежей к ненаблюдавшимся, но похожим. Обзор и эмпирические сравнения см. в работе See Chen and Goodman (1999). Одна из основных техник – прибавить ненулевую массу вероятности ко всем возможным значениям символов. Этот метод можно обосновать как байесовский вывод, в котором априорное распределение счетчиков равномерно или является распределением Дирихле. Еще одна очень популярная идея – образовать смесовую модель из n-граммных моделей высокого и низкого порядков, где модели высокого порядка обеспечивают бóльшую емкость, а модели низкого порядка с большей вероятностью избегают нулевых счетчиков. **Возвратные методы** ищут n-граммы низкого порядка, если частота контекста $x_{t-1}, \ldots, x_{t-n+1}$ слишком мала для использования модели более высокого порядка. Точнее, они оценивают распределение x_t с использованием контекстов $x_{t-n+k}, \ldots, x_{t-1}$ для возрастающих k, пока не будет найдена достаточно надежная оценка.

Классические n-граммные модели особенно уязвимы к проклятию размерности. Существует $|\mathbb{V}|^n$ возможных n-грамм, и $|\mathbb{V}|$ зачастую очень велико. Даже при наличии массивного обучающего набора и умеренном n большинство n-грамм в обучающем наборе не встречается. Классическую n-граммную модель можно рассматривать как поиск ближайшего соседа, т. е. как локальный непараметрический предиктор, похожий на метод k ближайших соседей. Статистические проблемы, присущие таким экстремально локальным предикторам, описаны в разделе 5.11.2. Но в языковой модели проблема даже серьезнее, чем обычно, потому что два любых разных слова находятся

на одинаковом расстоянии друг от друга в пространстве унитарных векторов. Поэтому трудно извлечь хоть какую-то информацию из «соседей» – лишь обучающие примеры, буквально повторяющие один и тот же контекст, полезны для локального обобщения. Для преодоления этих проблем модель языка должна уметь разделять знания между одним словом и другими семантически похожими словами.

Чтобы улучшить статистическую эффективность n-граммных моделей, в **классовые языковые модели** (Brown et al., 1992; Ney and Kneser, 1993; Niesler et al., 1998) введено понятие категории слова, и статистическая сила разделяется между словами из одной категории. Идея в том, чтобы применить алгоритм кластеризации для разбиения слов по кластерам, или классам, исходя из частоты совместной встречаемости с другими словами. После этого модель может использовать идентификаторы классов, а не идентификаторы отдельных слов для представления контекста справа от вертикальной черты в выражении условной вероятности. Возможны также составные модели, в которые словесные и классовые модели комбинируются путем смешения или перебора с возвратом. Хотя классы слов допускают обобщение на последовательности, в которых одно слово заменено другим из того же класса, при таком представлении теряется много информации.

12.4.2. Нейронные языковые модели

Нейронные языковые модели (НЯМ, англ. NLM) предназначены для преодоления проклятия размерности при моделировании последовательностей слов естественного языка посредством распределенного представления слов (Bengio et al., 2001). В отличие от классовых n-граммных моделей, нейронные языковые модели способны понять, что два слова похожи, не жертвуя возможностью кодировать каждое слово независимо от остальных. В нейронных языковых моделях статистическая сила разделяется между одним словом (и его контекстом) и другими похожими словами и контекстами. Распределенное представление, которому модель обучается для каждого слова, обеспечивает такое разделение, позволяя модели обрабатывать слова с общими признаками схожим образом. Например, если слова `dog` (собака) и `cat` (кошка) отображаются в представления, имеющие много общих атрибутов, то предложения, содержащие слово `cat`, могут влиять на предсказания, которые модель дает для предложений со словом `dog`, и наоборот. Поскольку таких атрибутов много, существует много способов обобщения, т. е. информация из каждого обучающего предложения переносится на экспоненциально большое число семантически родственных предложений. Проклятие размерности требует, чтобы количество предложений, на которые обобщается модель, экспоненциально зависело от длины предложения. Модель противостоит проклятию, сопоставляя каждому обучающему предложению экспоненциально большое число похожих предложений.

Иногда такие представления слов называются **векторными представлениями**, или **погружениями слов** (word embedding). При такой интерпретации мы рассматриваем исходные символы как точки в пространстве с числом измерений, равным размеру словаря. А векторное представление погружает эти точки в пространство признаков меньшей размерности. В исходном пространстве каждое слово представляется унитарным вектором, т. е. евклидово расстояние между любой парой слов равно $\sqrt{2}$. В пространстве признаков близки слова, часто встречающиеся в похожих контекстах (или любая пара слов с общими «признаками», которым обучилась модель). Часто это приводит к тому, что слова с похожим смыслом оказываются соседями. На рис. 12.3

приведены некоторые области пространства обученных признаков слов, чтобы показать, как семантически похожие слова отображаются в близкие представления.

Нейронные сети в других предметных областях также определяют погружения. Например, скрытый слой сверточной сети определяет «погружение изображения». Но для специалистов по ОЕЯ идея погружения представляет особый интерес, потому что изначально естественный язык не лежит в вещественном векторном пространстве. Скрытый слой дает качественно иное представление данных, существенно отличающееся от исходного.

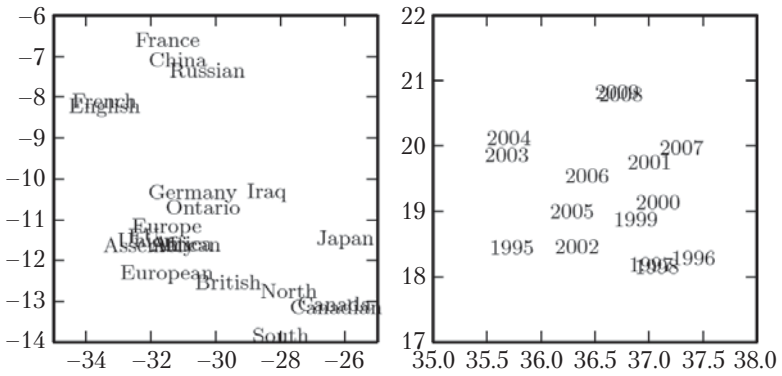

Рис. 12.3 ❖ Двумерная визуализация погружения слов, полученных от нейронной модели машинного перевода (Bahdanau et al., 2015). Выделены области, где векторы, соответствующие семантически родственным словам, близки друг к другу. Слева показаны страны, справа – числа. Не забывайте, что двумерное векторное представление показано только для наглядности. В реальных приложениях размерность векторного представления обычно выше, поскольку одновременно улавливаются различные аспекты сходства между словами

Идея использования распределенных представлений для улучшения моделей обработки естественных языков не ограничивается нейронными сетями. Ее можно распространить и на графические модели, имеющие распределенные представления в форме нескольких скрытых переменных (Mnih and Hinton, 2007).

12.4.3. Многомерные выходы

Во многих приложениях для обработки естественных языков мы хотим, чтобы модель выводила слова, а не символы. Если словарь большой, то вычислительно очень трудно представить выходное распределение выбора слов. Часто словарь $\mathbb{V}$ насчитывает сотни тысяч слов. Наивный подход к представлению такого распределения – выполнить аффинное преобразование скрытого представления на пространство выходов, а затем применить функцию softmax. Пусть имеется словарь $\mathbb{V}$ размера $|\mathbb{V}|$. Матрица весов, описывающая линейную составляющую такого аффинного преобразования, очень велика, поскольку размерность результата составляет $|\mathbb{V}|$. Поэтому для ее хранения требуется много памяти, а умножение на нее займет много времени. Поскольку softmax нормируется по всем $|\mathbb{V}|$ выходам, придется выполнять полное умножение на матрицу как на этапе обучения, так и на этапе тестирования – мы не можем ограничиться только вычислением скалярного произведения на вектор весов. Таким

образом, вычислительная стоимость выходного слоя велика как во время обучения (нужно вычислять правдоподобие и его градиент), так и во время тестирования (нужно вычислять вероятности всех или избранных слов). Для специальных функций потерь градиент можно вычислить эффективно (Vincent et al., 2015), но применение стандартной перекрестной энтропии к традиционному выходному слою с функцией softmax сталкивается с многочисленными трудностями.

Предположим, что $\boldsymbol{h}$ – верхний скрытый слой, используемый для предсказания вероятностей выходов $\hat{\boldsymbol{y}}$. Если параметризовать преобразование $\boldsymbol{h}$ в $\hat{\boldsymbol{y}}$ обученными весами $\boldsymbol{W}$ и смещениями $\boldsymbol{b}$, то выходной слой с аффинным преобразованием и функцией softmax выполняет следующие вычисления:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \ldots, |\mathbb{V}|\}; \tag{12.8}$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}. \tag{12.9}$$

Если $\boldsymbol{h}$ содержит n_h элементов, то сложность этой операции равна $O(|\mathbb{V}|n_h)$. Когда n_h порядка тысячи, а $|\mathbb{V}|$ порядка сотен тысяч, на эту операцию приходится основное время вычислений в большинстве моделей на основе нейронных сетей.

12.4.3.1. Использование короткого списка

В первых нейронных языковых моделях (Bengio et al., 2001, 2003) проблема высокой стоимости использования softmax при большом числе выходных слов решалась ограничением размера словаря 10–20 тысячами слов. В работах Schwenk and Gauvain (2002) и Schwenk (2007) этот подход подвергся дальнейшему развитию: словарь $\mathbb{V}$ был разделен на **короткий список** $\mathbb{L}$ самых частых слов (обрабатываемый нейронной сетью) и хвост $\mathbb{T} = \mathbb{V} \backslash \mathbb{L}$ более редких слов (обрабатываемый n-граммной моделью). Чтобы оба предсказания можно было объединить, нейронная сеть должна также предсказывать вероятность, что слово, следующее за контекстом C, принадлежит хвостовому списку. Для этого можно добавить дополнительный сигмоидный выходной блок, дающий оценку $P(i \in \mathbb{T} \mid C)$. Тогда дополнительный выход можно использовать для получения оценки распределения вероятности всех слов из $\mathbb{V}$:

$$\begin{aligned} P(y = i \mid C) = {} & 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L})(1 - P(i \in \mathbb{T} \mid C)) \\ & + 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C), \end{aligned} \tag{12.10}$$

где $P(y = i \mid C, i \in \mathbb{L})$ порождается нейронной языковой моделью, а $P(y = i \mid C, i \in \mathbb{T})$ – n-граммной моделью. После небольшой модификации этот подход будет работать, если ввести дополнительное выходное значение в softmax-слой нейронной языковой модели, а не заводить отдельный сигмоидный блок.

Очевидный недостаток короткого списка состоит в том, что потенциальный выигрыш от обобщаемости нейронных языковых моделей ограничен самыми часто встречающимися словами, а они-то как раз наименее полезны. Этот недостаток стимулировал интерес к описанным ниже альтернативным методам работы с многомерным выходным слоем.

12.4.3.2. Иерархическое вычисление softmax

Классический подход (Goodman, 2001) к уменьшению вычислительной нагрузки на многомерные выходные слои при большом размере словаря $\mathbb{V}$ – иерархическое раз-

ложение вероятностей. Объем вычислений, пропорциональный $|\mathbb{V}|$ (и количеству скрытых слоев n_h), можно сократить до величины, пропорциональной $\log|\mathbb{V}|$. В работах Bengio (2002) и Morin and Bengio (2005) эта идея факторизации применена в контексте нейронных языковых моделей.

Иерархию можно представлять себе как построение категорий слов, затем категорий категорий слов и т. д. Вложенные категории образуют дерево, в листьях которого находятся слова. Глубина сбалансированного дерева равна $O(\log|\mathbb{V}|)$. Вероятность выбора слова равна произведению вероятностей выбора ветви на всем пути, ведущем от корня к этому слову. На рис. 12.4 приведен простой пример. В работе Mnih and Hinton (2009) описано также, как использовать несколько путей для идентификации одного слова, чтобы лучше смоделировать слова, имеющие несколько значений. Тогда вычисление вероятности слова сводится к суммированию по всем ведущим к нему путям.

Для предсказания условных вероятностей, необходимых в каждом узле дерева, мы обычно используем модель логистической регрессии в узлах и подаем на вход всех моделей один и тот же контекст C. Поскольку правильный выход уже указан в обучающем наборе, для обучения моделей логистической регрессии можно применить обучение с учителем. Обычно при этом используется стандартная перекрестная энтропия в качестве функции потерь, что соответствует максимизации логарифмического правдоподобия правильной последовательности решений.

Поскольку выходное логарифмическое правдоподобие можно вычислить эффективно (объем вычислений пропорционален $\log|\mathbb{V}|$, а не $|\mathbb{V}|$), то и градиенты тоже вычисляются эффективно. И это относится не только к градиентам по выходным параметрам, но и к градиентам по активациям скрытых слоев.

Возможно, хотя на практике так обычно не поступают, оптимизировать структуру дерева, чтобы уменьшить ожидаемый объем вычислений. Методы теории информации говорят, как построить оптимальный двоичный код, если известны относительные частоты слов. Для этого мы могли бы структурировать дерево так, чтобы число бит, ассоциированных со словом, было приблизительно равно логарифму частоты этого слова. На практике, однако, такая экономия редко стоит усилий, потому что вычисление выходных вероятностей – это лишь часть общего объема вычислений в нейронной языковой модели. Предположим, к примеру, что имеется l полносвязных скрытых слоев ширины n_h. Обозначим n_b взвешенное среднее числа бит, необходимых для идентификации слова, в котором вес слова равен его частоте. В нашем случае число операций, выполняемых для вычисления скрытых активаций, растет как $O(ln_h^2)$, а объем вычислений выхода – как $O(n_h n_b)$. Если $n_b \leq ln_h$, то для сокращения объема вычислений лучше уменьшать n_h, а не n_b. На самом деле n_b часто мало. Поскольку размер словаря редко превышает миллион слов, а $\log_2(10^6) \approx 20$, можно уменьшить n_b примерно до 20, тогда как n_h обычно гораздо больше – порядка 10^3 и более. Вместо того чтобы тщательно оптимизировать дерево с коэффициентом ветвления 2, мы можем определить дерево глубины 2 с коэффициентом ветвления $\sqrt{|\mathbb{V}|}$. Такое дерево соответствует определению множества взаимно исключающих классов слов. Простой подход, основанный на дереве глубины 2, сохраняет большую часть вычислительных преимуществ иерархической стратегии.

Остается открытым один вопрос – как лучше всего определить классы слов, или как определить иерархию слов вообще. В ранних работах использовались существующие иерархии (Morin and Bengio, 2005), однако иерархию тоже можно обучить, в иде-

але совместно с нейронной языковой моделью. Обучать иерархию трудно. Точная оптимизация логарифмического правдоподобия представляется невозможной, потому что выбор иерархии слов – дискретная задача, к которой градиентная оптимизация неприменима. Однако можно воспользоваться дискретной оптимизацией, чтобы приблизительно аппроксимировать разбиение слов по классам.

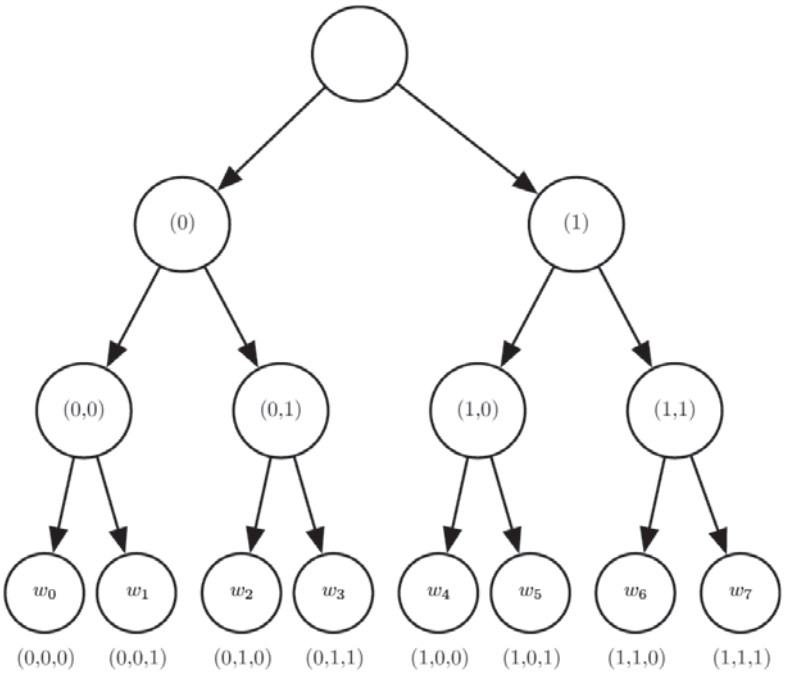

Рис. 12.4 ❖ Иллюстрация простой иерархии категорий слов на примере 8 слов w_0, ..., w_7, организованных в трехуровневую иерархию. В листьях дерева находятся сами слова. Внутренние узлы представляют группы слов. Каждый узел можно индексировать последовательностью бинарных решений (0 = влево, 1 = вправо), описывающих путь от корня к этому узлу. Суперкласс (0) содержит два класса: (0, 0) и (0, 1), содержащих соответственно множества слов $\{w_0, w_1\}$ и $\{w_2, w_3\}$. Аналогично суперкласс (1) содержит два класса: (1, 0) и (1, 1), содержащих соответственно множества слов $\{w_4, w_5\}$ и $\{w_6, w_7\}$. Если дерево достаточно сбалансировано, то максимальная глубина (число бинарных решений) по порядку величины равна логарифму числа слов $|\mathbb{V}|$: для выбора одного из $|\mathbb{V}|$ слов нужно произвести $O(\log|\mathbb{V}|)$ операций (по одной для каждого узла на пути от корня). В этом примере для вычисления вероятности слова *y* нужно перемножить три вероятности, ассоциированные с бинарными решениями, принимаемыми в каждом узле на пути от корня к узлу y. Обозначим $b_i(y)$ – i-е бинарное решение, принимаемое в процессе обхода дерева на пути к значению y. Тогда вероятность выборки выхода y разлагается в произведение условных вероятностей. Например, вероятность w_4 можно представить в виде произведения следующим образом:

$$P(\mathrm{y} = w_4) = P(\mathrm{b}_0 = 1, \mathrm{b}_1 = 0, \mathrm{b}_2 = 0) \tag{12.11}$$

$$= P(\mathrm{b}_0 = 1)P(\mathrm{b}_1 = 0 \,|\, \mathrm{b}_0 = 1)P(\mathrm{b}_2 = 0 \,|\, \mathrm{b}_0 = 1, \mathrm{b}_1 = 0). \tag{12.12}$$

Важное преимущество иерархического вычисления softmax – тот факт, что вычислительный выигрыш достигается как на этапе обучения, так и на этапе тестирования, если во время тестирования мы захотим вычислить вероятности конкретных слов.

Разумеется, вычисление вероятностей всех $|\mathbb{V}|$ слов по-прежнему обходится дорого, даже при иерархическом подходе. Еще одна важная операция – выбор самого вероятного слова в данном контексте. К сожалению, древовидная структура не дает эффективного и точного решения этой задачи.

Недостаток состоит в том, что на практике иерархическое вычисление softmax часто дает при тестировании худшие результаты, чем выборочные методы, описанные ниже. Это может быть связано с неудачным выбором классов слов.

12.4.3.3. Выборка по значимости

Ускорить обучение нейронных языковых моделей можно, в частности, избежав явного вычисления вклада в градиент от всех слов, которые не встречаются в следующей позиции. У любого недопустимого слова должна быть низкая вероятность в модели. Перечисление всех таких слов стоит дорого. Но можно вместо этого выбрать лишь их подмножество. В обозначениях из формулы (12.8) градиент можно записать в следующем виде:

$$\frac{\partial \log P(y|C)}{\partial \theta} = \frac{\partial \log \text{soft}\max_y(\boldsymbol{a})}{\partial \theta} \tag{12.13}$$

$$= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \tag{12.14}$$

$$= \frac{\partial}{\partial \theta} \left(a_y - \log \sum_i e^{a_i} \right) \tag{12.15}$$

$$= \frac{\partial a_y}{\partial \theta} - \sum_i P(y=i|C) \frac{\partial a_i}{\partial \theta}, \tag{12.16}$$

где $\boldsymbol{a}$ – вектор активаций до применения softmax (оценок), по одному элементу на каждое слово. Первый член – **положительная фаза**, толкающая a_y вверх, второй член – **отрицательная фаза**, толкающая a_i вниз с весом $P(i \mid C)$ для всех i. Поскольку отрицательная фаза – это математическое ожидание, то оценить ее можно по выборке Монте-Карло. Однако для этого потребовалось бы производить выборку из самой модели. Выборка из модели подразумевает вычисление $P(i \mid C)$ для всех i в словаре, а это как раз то, чего мы хотим избежать.

Вместо выборки из модели мы можем произвести выборку из другого распределения, которое называется вспомогательным распределением (proposal distribution) и обозначается q, и использовать подходящие веса для корректировки смещения вследствие выборки не из того распределения (Bengio and Sénécal, 2003; Bengio and Sénécal, 2008). Это пример применения более общей техники, называемой **выборкой по значимости**, подробно она будет описана в разделе 17.2. К сожалению, даже точная выборка по значимости не эффективна, потому что требуется вычисление весов p_i/q_i, где $p_i = P(i \mid C)$, а их можно вычислить, только если вычислены все веса a_i. Специально для данного приложения предложено решение, называемое **смещенной выборкой по значимости**, когда веса нормированы, так что их сумма равна 1. Если выбрано отрицательное слово n_i, то ассоциированный градиент входит с весом

$$w_i = \frac{p_{n_i} / q_{n_i}}{\sum_{j=1}^{N} p_{n_j} / q_{n_j}}. \tag{12.17}$$

С помощью этих весов придается подходящая значимость m отрицательным примерам из q, используемым для формирования оценки вкладка отрицательной фазы в градиент:

$$\sum_{i=1}^{|\mathbb{V}|} P(i|C) \frac{\partial a_i}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^{m} w_i \frac{\partial a_{n_i}}{\partial \theta}. \tag{12.18}$$

В качестве вспомогательного распределения q вполне можно использовать распределение униграмм или биграмм. Параметры такого распределения легко оценить по данным. После оценки параметров выборка из распределения производится очень эффективно. Выборка по значимости полезна не только для ускорения работы моделей с большими выходными softmax-слоями, но и во всех случаях, когда нужно ускорить обучение при наличии большого выходного слоя, представленного разреженным вектором, а не выбором 1 из n. Примером может служить **набор слов** (bag of words). Это разреженный вектор $\boldsymbol{v}$, элемент v_i которого обозначает присутствие или отсутствие в документе i-го слова из словаря. Альтернативно v_i может показывать, сколько раз встречается i-е слово. Обучить модель машинного обучения, порождающую такие векторы, бывает трудно по ряду причин. На ранних стадиях обучения модель не всегда порождает по-настоящему разреженный выход. Кроме того, используемая при обучении функция потерь, возможно, более естественно описывается в терминах сравнения каждого элемента выхода с меткой. Это означает, что не всегда очевидно, есть ли вычислительный выигрыш от использования разреженного выхода, поскольку модель может сделать большинство выходных элементов ненулевыми, и все эти ненулевые элементы придется сравнивать с соответствующими обучающими метками, даже если метка нулевая. В работе Dauphin et al. (2011) продемонстрировано, что такие модели можно ускорить с помощью выборки по значимости. Эффективный алгоритм минимизирует реконструкцию потери для «положительных слов» (для которых метка ненулевая) и равного числа «отрицательных слов». Отрицательные слова выбираются случайным образом с применением эвристики для выбора слов, которые с большей вероятностью будут ошибочными. Смещение, вызванное такой выборкой с запасом, можно затем скорректировать с помощью весов значимости.

Во всех этих случаях вычислительная сложность оценивания градиента для выходного слоя уменьшается и становится пропорциональной числу отрицательных примеров, а не размеру выходного вектора.

12.4.3.4. Шумосопоставительное оценивание и потеря ранжирования

Были предложены и другие способы уменьшения вычислительной сложности обучения нейронных языковых моделей с большими словарями. Один из ранних подходов – потеря ранжирования (ranking loss) – описан в работе Collobert and Weston (2008a), где выход нейронной языковой модели для каждого слова рассматривается как балльная оценка и производится попытка сделать так, чтобы оценка правильного слова a_y ранжировалась выше, по сравнению с остальными оценками a_i. Тогда потеря ранжирования равна

$$L = \sum_i \max(0, 1 - a_y + a_i). \tag{12.19}$$

Градиент для i-го члена равен нулю, если оценка наблюдаемого слова a_y больше оценки отрицательного слова a_i не менее, чем на 1. У этого критерия есть недостаток – он не дает оценку условных вероятностей, что полезно в некоторых приложениях, в т. ч. для распознавания речи и порождения текста (включая условное порождение текста, как в случае перевода).

Позже в качестве целевой функции обучения для нейронных языковых моделей было предложено шумосопоставительное оценивание (noise-contrastive estimation), описанное в разделе 18.6 (Mnih and Teh, 2012; Mnih and Kavukcuoglu, 2013).

12.4.4. Комбинирование нейронных языковых моделей с *n*-граммами

Важное преимущество n-граммных моделей, по сравнению с нейронными сетями, состоит в том, что первые достигают высокой емкости (за счет хранения частот очень большого числа кортежей) при очень скромном объеме вычислений в ходе обработки примера (требуется найти лишь немного примеров, соответствующих текущему контексту). Если для доступа к счетчикам использовать хэш-таблицы или деревья, то объем вычислений в n-граммной модели почти не зависит от емкости. Для сравнения – удвоение числа параметров нейронной сети обычно приводит к увеличению времени вычислений примерно вдвое. Исключение составляют модели, в которых на каждом проходе используются не все параметры. При наличии слоев погружения на каждом проходе индексируется только одно погружение, поэтому размер словаря можно увеличить, не увеличивая времени обработки каждого примера. В некоторых других моделях, например периодических сверточных сетях, можно добавлять параметры, одновременно уменьшая степень разделения параметров, чтобы сохранить объем вычислений на прежнем уровне. Но в слоях типичных нейронных сетей, основанных на умножении матриц, объем вычислений пропорционален числу параметров.

Таким образом, для увеличения емкости можно поступить просто: построить ансамбль, содержащий нейронную и n-граммную языковую модель (Bengio et al., 2001, 2003). Как и любая ансамблевая техника, этот метод может уменьшить ошибку тестирования, если члены ансамбля совершают ошибки независимо друг от друга. В ансамблевом обучении есть много способов скомбинировать предсказания отдельных членов, в т. ч. равномерное взвешивание и выбор весов на контрольном наборе. В работе Mikolov et al. (2011a) ансамбль обобщен с двух моделей на большой массив моделей. Можно также объединить нейронную сеть с моделью максимальной энтропии и обучить обе совместно (Mikolov et al., 2011b). Этот подход можно рассматривать как обучение нейронной сети с дополнительным множеством входов, напрямую связанных с выходом и не связанных ни с какой другой частью модели. Дополнительные входы указывают на присутствие определенных n-грамм во входном контексте, так что эти переменные имеют очень высокую размерность и сильно разрежены. Увеличение емкости модели получается гигантским – новая часть архитектуры содержит до $|sV|^n$ параметров, но дополнительный объем вычислений минимален, потому что добавленные входные данные крайне разрежены.

12.4.5. Нейронный машинный перевод

Задача машинного перевода состоит в чтении предложения на одном естественном языке и выводе предложения с эквивалентным смыслом на другом языке. Системы машинного перевода обычно состоят из многих компонент. На верхнем уровне име-

ется один компонент, который предлагает большое число потенциальных переводов. Многие переводы грамматически некорректны из-за различий между языками. Например, в некоторых языках прилагательные ставятся после существительных, так что при переводе на английский получаются фразы типа «apple red». Механизм предложений рекомендует много вариантов перевода, и в идеале среди них будет и «red apple». Второй компонент системы перевода, языковая модель, оценивает предложенные переводы и может поставить варианту «red apple» более высокую оценку, чем варианту «apple red».

Уже в самых первых исследованиях применения нейронных сетей к машинному переводу встречалась идея кодировщика и декодера (Allen 1987; Chrisman 1991; Forcada and Ñeco 1997), а первым крупным коммерческим приложением в этой области стал переход на нейронную языковую модель в одной системе перевода (Schwenk et al., 2006; Schwenk, 2010). Ранее в большинстве систем машинного перевода для этой цели применялась n-граммная модель. В машинном переводе используются не только традиционные возвратные n-граммные модели (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999), но также **языковые модели максимальной энтропии** (Berger et al., 1996), в которых слой с аффинным преобразованием и функцией softmax предсказывает следующее слово при наличии частых n-грамм в контексте.

Традиционные языковые модели просто возвращают вероятность предложения естественного языка. Поскольку задача машинного перевода – породить выходное предложение по известному входному, имеет смысл обобщить модель естественного языка, сделав ее условной. В разделе 6.2.1.1 был описан прямолинейный способ обобщения модели, определяющей маргинальное распределение некоторой переменной таким образом, чтобы она определяла условное распределение той же переменной при заданном контексте C, где C может быть одной переменной или списком переменных. В работе Devlin et al. (2014) превзойдено лучшее достижение на некоторых эталонных тестах для машинного перевода; для этого авторы использовали МСП, который оценивал фразу $t_1, t_2, ..., t_k$ на целевом языке, зная фразу $s_1, s_2, ..., s_n$ на исходном языке. МСП вычисляет оценку в виде вероятности $P(t_1, t_2, ..., t_k \mid s_1, s_2, ..., s_n)$. Оценка, вычисленная этим МСП, используется вместо оценки, предложенной условной n-граммной моделью.

Недостаток подхода на основе МСП состоит в том, что последовательности нужно предварительно обработать, так чтобы их длина была фиксирована. Для повышения гибкости перевода хотелось бы иметь модель, которая подстраивается под входы и выходы переменной длины. Такую возможность дает рекуррентная нейронная сеть. В разделе 10.2.4 описано несколько способов построения РНС, представляющей условное распределение последовательности при условии входа, а в разделе 10.4 – как реализовать такое обусловливание, когда входом является последовательность. В любом случае сначала одна модель читает входную последовательность и порождает структуру данных, содержащую ее сводку. Эта сводка называется «контекстом» C. Контекст может быть списком векторов, вектором или тензором. Модель, которая читает вход и порождает C, может представлять собой РНС (Cho et al., 2014a; Sutskever et al., 2014; Jean et al., 2014) или сверточную сеть (Kalchbrenner and Blunsom, 2013). Затем вторая модель, обычно РНС, читает контекст C и порождает предложение на целевом языке. Эта общая идея кодировщика-декодера для машинного перевода показана на рис. 12.5.

Рис. 12.5 ❖ Архитектура кодировщик-декодер, реализующая отображение между поверхностным представлением (например, последовательностью слов или изображением) и семантическим представлением. Кодировщик преобразует данные, представленные в одной модальности (например, предложение на французском языке в скрытое представление, улавливающее смысл предложения), и его выход подается на вход декодера, ориентированного на другую модальность (например, преобразующего скрытое семантическое представление в предложение на английском языке). Применяя кодировщик и декодер, мы можем обучить систему для перевода из одной модальности в другую. Эта идея с успехом применялась не только к машинному переводу, но и для генерации подписей к изображениям

Чтобы сгенерировать все предложение, обусловленное предложением на исходном языке, модель должна как-то представить целое исходное предложение. Ранние модели умели представлять только отдельные слова или фразы. Было бы полезно обучить такое представление, что предложения с одинаковым смыслом имеют схожие представления и на исходном, и на целевом языке. Сначала была предпринята попытка реализовать такую стратегию с использованием комбинации свертки с РНС (Kalchbrenner and Blunsom, 2013). Позже РНС стала применяться для оценки предложенных переводов (Cho et al., 2014a) и порождения переведенных предложений (Sutskever et al., 2014). В работе Jean et al. (2014) эти модели масштабированы на словари большего размера.

12.4.5.1. Использование механизма внимания и совмещение частей данных

Уловить все семантические детали очень длинного предложения, скажем из 60 слов, в представлении фиксированного размера крайне трудно. Этого можно добиться, обучая достаточно большую РНС в течение достаточно длительного времени, как показано в работах Cho et al. (2014a) и Sutskever et al. (2014). Но есть и более эффективный подход: прочитать все предложение или абзац (чтобы получить контекст и в общих чертах понять, о чем речь), а затем порождать переводы слов по одному, всякий раз фокусируясь на новой части входного предложения, чтобы собрать семантические детали, необходимые для порождения следующего слова. Эта идея впервые была реализована в работе Bahdanau et al. (2015). Механизм внимания, использо-

ванный для фокусирования на отдельных частях входного предложения на каждом временном шаге, иллюстрируется на рис. 12.6.

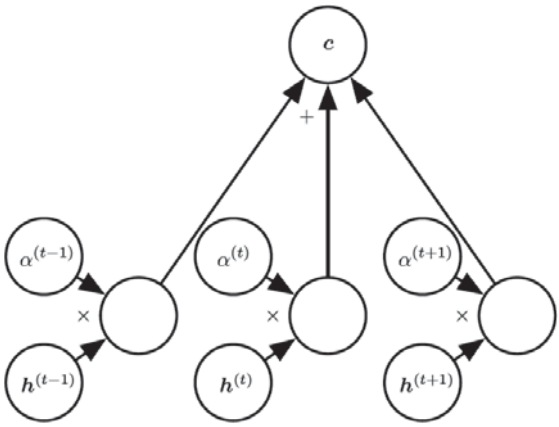

Рис. 12.6 ❖ Современный механизм внимания, введенный в работе Bahdanau et al. (2015), по существу, представляет собой взвешенное среднее. Вектор контекста $\boldsymbol{c}$ образуется путем вычисления взвешенного среднего векторов признаков $\boldsymbol{h}^{(t)}$ с весами $\alpha^{(t)}$. В некоторых приложениях векторы признаков $\boldsymbol{h}$ – скрытые блоки нейронной сети, но это могут быть и исходные данные модели. Веса $\alpha^{(t)}$ порождает сама модель. Обычно это значения из отрезка [0, 1], которые концентрируются вокруг единственного значения $\boldsymbol{h}^{(t)}$, чтобы взвешенное среднее аппроксимировало чтение именно на этом временном шаге. Как правило, веса $\alpha^{(t)}$ являются результатом применения функции softmax к оценкам релевантности, вычисленным другой частью модели. Вычислительно механизм внимания дороже прямого индексирования желаемого $\boldsymbol{h}^{(t)}$, но прямому индексированию невозможно обучиться методом градиентного спуска. Механизм внимания, основанный на взвешенных средних, – гладкая дифференцируемая аппроксимация, допускающая обучение существующими алгоритмами оптимизации

Можно считать, что система с механизмом внимания состоит из трех компонентов:

1) процесс, который *читает* исходные данные (например, слова исходного предложения) и преобразует их в распределенное представление, ассоциируя один вектор признаков с каждой позицией слова;
2) список векторов признаков, построенный читателем. Его можно трактовать как *память*, содержащую последовательность фактов, которые можно впоследствии извлекать, необязательно в том же порядке и необязательно перебирая все;
3) процесс, который последовательно выполняет некоторую задачу, *обращаясь* к содержимому памяти. На каждом временном шаге у него есть возможность акцентировать внимание на содержимом одного элемента памяти (или нескольких, с разными весами).

Третий компонент порождает переведенное предложение.

Когда слова предложения, написанного на одном языке, совмещаются с соответственными словами переведенного предложения, становится возможным сопоставить соответствующие погружения слов. В более ранней работе показано, как обучить своего рода матрицу перевода, сопоставляющую погружения слов на разных языках

(Kociský et al., 2014), получив при этом меньшую частоту ошибок совмещения, чем в традиционных решениях, основных на подсчете частот в таблице фраз. Существует и еще более ранняя работа по обучению межъязыковых векторов слов (Klementiev et al., 2012). Этот подход можно развить в разных направлениях. Например, более эффективное межъязыковое совмещение (Gouws et al., 2014) позволяет проводить обучение на бóльших наборах данных.

12.4.6. Историческая справка

Идея распределенных представлений символов впервые была высказана в работе Rumelhart et al. (1986a) – одном из первых исследований обратного распространения; символы там соответствовали идентификаторам членов семьи, нейронная сеть улавливала связи между членами семьи, а обучающие примеры представлялись тройками вида (Колин, мать, Виктория). Первый слой сети обучался представлению каждого члена семьи. Например, для Колина могли быть выделены такие признаки: в каком генеалогическом древе он находится, в какой ветви этого древа, в каком поколении и т. д. Можно считать, что нейронная сеть вычисляет правила, связывающие эти атрибуты для получения желаемых предсказаний. Обученная сеть может, например, вывести, кто приходится матерью Колину.

Идея формирования погружения символа была обобщена на идею погружения слова в работе Deerwester et al. (1990). Для обучения погружений использовалось спектральное разложение. Позже для этой цели применили бы нейронные сети. История обработки естественных языков отмечена сменой популярности различных представлений входа модели. По следам этой ранней работы по символам и словам в первых приложениях нейронных сетей к ОЕЯ (Miikkulainen and Dyer, 1991; Schmidhuber, 1996) вход представлялся в виде последовательности литер.

В работе Bengio et al. (2001) произошел возврат к моделированию слов и были введены нейронные языковые модели, порождающие интерпретируемые погружения слов. Эти модели постепенно масштабировались: от представлений небольшого набора символов в 1980-х годах до миллионов слов (включая имена собственные и неправильные написания) в современных приложениях. Усилия, направленные на достижение вычислительной масштабируемости, привели к изобретению техник, описанных в разделе 12.4.3.

В самом начале использование слов в качестве фундаментальных единиц языка привело к повышению качества языкового моделирования (Bengio et al., 2001). Сегодня появились новые методы и для моделей на основе литер (Sutskever et al., 2011), и для моделей на основе слов (Gillick et al., 2015), и даже для моделирования отдельных байтов литер в кодировке Unicode.

Идеи, стоящие за нейронными языковыми моделями, были распространены и на другие приложения ОЕЯ, в т. ч. грамматический разбор (Henderson, 2003, 2004; Collobert, 2011), частеречная разметка, пометка семантических ролей, фрагментация (chunking) и т. д. Иногда при этом применяется единая многозадачная архитектура обучения (Collobert and Weston, 2008a; Collobert et al., 2011a), в которой погружения слов сообща используются разными задачами.

Двумерная визуализация погружений стала популярным инструментом анализа языковых моделей после разработки алгоритма понижения размерности t-SNE (van der Maaten and Hinton, 2008) и его широко известного применения к задаче визуализации погружений слов, предложенного Джозефом Турианом в 2009 году.

12.5. Другие приложения

В этом разделе мы рассмотрим еще несколько приложений глубокого обучения, помимо описанных выше стандартных задач распознавания объектов, распознавания речи и обработки естественных языков. В третьей части книги мы добавим к этому перечню задачи, пока еще не вышедшие из стадии исследования.

12.5.1. Рекомендательные системы

Одно из основных семейств приложений машинного обучения к информационным технологиям – возможность давать рекомендации потенциальным пользователям или заказчикам товаров и услуг. Можно выделить два типа приложений: онлайновая реклама и рекомендация продуктов (зачастую рекомендации преследуют ту же цель: продать продукт). В обоих случаях требуется предсказать ассоциацию между пользователем и продуктом – для того чтобы предсказать либо вероятность некоторого действия (покупки продукта или какого-то эквивалента этому действию), либо ожидаемую выгоду (которая может зависеть от ценности продукта) от показа рекламного объявления или рекомендации продукта пользователю. В настоящее время Интернет финансируется в значительной мере за счет различных видов онлайновой рекламы. Многие отрасли экономики зависят от покупок через Интернет. Такие компании, как Amazon и eBay, применяют машинное обучение, в т. ч. глубокое, для рекомендования своих товаров и услуг. Иногда речь вообще не идет о продаже чего-либо. В качестве примеров можно назвать отбор сообщений, отображаемых в новостной ленте социальной сети, рекомендование фильмов, анекдотов, советов специалистов, подбор партнеров для видеоигр или паросочетания в сервисах знакомств.

Часто проблема ассоциации рассматривается как задача обучения с учителем: зная какую-то информацию о продукте и пользователе, предсказать тот или иной вариант выражения заинтересованности (переход пользователем по ссылке, ввод рейтинга, нажатие на кнопку «Нравится», покупка продукта, трата какой-то суммы денег на продукт, переход на страницу с описанием продукта и т. д.). Часто все сводится либо к задаче регрессии (предсказать ожидаемое значение при некоторых условиях), либо к задаче вероятностной классификации (предсказать условную вероятность дискретного события).

Ранние работы по рекомендательным системам опирались на минимальную входную информацию для предсказаний: идентификатор пользователя и идентификатор продукта. В этом контексте для обобщения можно полагаться только на сходство между паттернами значений целевой переменной для разных пользователей или разных продуктов. Предположим, что пользователям 1 и 2 нравятся продукты A, B и C. Отсюда можно сделать вывод, что у пользователей 1 и 2 схожие вкусы. Если пользователю 1 нравится продукт D, то велики шансы, что он понравится и пользователю 2. Алгоритмы, основанные на этой идее, называются **коллаборативной фильтрацией**. Возможны как непараметрические подходы (метод ближайших соседей, основанный на оценке сходства между паттернами предпочтений), так и параметрические. Параметрические методы зачастую опираются на обучение распределенного представления (называемого также погружением) для каждого пользователя и каждого продукта. Билинейное предсказание целевой переменной (например, рейтинга) – простой параметрический метод, оказавшийся чрезвычайно успешным и часто встречающийся в самых передовых системах в качестве одного из компонентов. Для предсказа-

ния вычисляется скалярное произведение погружения пользователя и погружения продукта (возможно, скорректированное с помощью констант, зависящих только от идентификатора пользователя или продукта). Обозначим $\hat{\boldsymbol{R}}$ матрицу наших предсказаний, $\boldsymbol{A}$ – матрицу, в которой по строкам расположены погружения пользователей, а $\boldsymbol{B}$ – матрицу, в котором по столбцам расположены погружения объектов. Пусть $\boldsymbol{b}$ и $\boldsymbol{c}$ – векторы, содержащие соответственно разновидности смещения для пользователей (насколько пользователь брюзгливый или жизнерадостный) и для продуктов (общая популярность продукта). Тогда билинейное предсказание имеет вид:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}. \tag{12.20}$$

Обычно стремятся минимизировать квадратичную ошибку между предсказанными рейтингами $\hat{R}_{u,i}$ и фактическими рейтингами $R_{u,i}$. Погружения пользователей и продуктов можно удобно визуализировать, предварительно понизив размерность (до 2 или 3), или использовать для сравнения пользователей или продуктов между собой так же, как это делается для погружений слов. Один из способов получить погружения – вычислить сингулярное разложение матрицы $\boldsymbol{R}$ фактических целей (например, рейтингов). Это соответствует разложению $\boldsymbol{R} = \boldsymbol{UDV'}$ (или нормированного варианта) в произведение двух множителей: матриц низкого ранга $\boldsymbol{A} = \boldsymbol{UD}$ и $\boldsymbol{B} = \boldsymbol{V'}$. Проблема в том, что в сингулярном разложении отсутствующие элементы произвольным образом трактуются так, будто они соответствуют целевому значению 0. А мы вообще не хотели бы платить за предсказания, сделанные на основе отсутствующих значений. К счастью, сумму квадратов ошибок на наблюдаемых рейтингах также легко минимизировать градиентными методами. Сингулярное разложение и билинейное предсказание (12.20) показали очень хорошие результаты в соревновании на приз компании Netflix (Bennett and Lanning, 2007), где требовалось предсказать рейтинги фильмов, зная только предыдущие рейтинги, выставленные большим числом анонимных пользователей. В этом соревновании, проходившем в период с 2006 по 2009 год, участвовали многие специалисты по машинному обучению. Оно стимулировало новые исследования по применению передовых методов машинного обучения к рекомендательным системам и в конечном итоге привело к улучшению последних. И хотя ни простое билинейное предсказание, ни сингулярное разложение не добились победы сами по себе, они были компонентами ансамблевых моделей, представленных большинством участников, в т. ч. победителями (Töscher et al., 2009; Koren, 2009).

Одно из первых применений нейронных сетей к коллаборативной фильтрации было основано на неориентированной вероятностной модели в виде ограниченной машины Больцмана (Salakhutdinov et al., 2007). ОМБ входила важной составной частью в ансамбль моделей, выигравший соревнование на приз Netflix (Töscher et al., 2009; Koren, 2009). В сообществе нейронных сетей исследовались и более продвинутые варианты идеи факторизации матрицы рейтингов (Salakhutdinov and Mnih, 2008).

Однако у систем коллаборативной фильтрации есть существенное ограничение: когда появляется новый пользователь или продукт, у него нет никакой истории рейтингования, а потому невозможно оценить его сходство с другими пользователями или продуктами или степень ассоциации между новым пользователем и существующими продуктами. Это проблема холодного старта. Общий способ ее решения – добавить информацию о пользователях и продуктах. Это может быть, например, профиль

пользователя или признаки продукта. Рекомендательные системы, в которых такая информация используется, называются **системами фильтрации по содержимому**. Отображение подробного набора признаков пользователей или продуктов на погружение можно обучить, применив архитектуру глубокого обучения (Huang et al., 2013; Elkahky et al., 2015).

Для выделения признаков из сложного содержимого, например из музыкальных треков для рекомендования музыки, также применялись специализированные архитектуры глубокого обучения, в частности сверточные сети (van den Oörd et al., 2013). В этой работе сверточная сеть принимала на входе акустические признаки и вычисляла векторное представление соответствующей песни. Затем скалярное произведение векторных представлений песни и пользователя использовалось, чтобы предсказать, будет пользователь слушать песню или нет.

12.5.1.1. Исследование и использование

В задаче выработки рекомендаций для пользователей возникает проблема, выходящая за рамки обычного обучения с учителем в плоскость обучения с подкреплением. Многие проблемы рекомендования теоретически точнее всего описываются как **контекстуальные бандиты** (Langford and Zhang, 2008; Lu et al., 2010). Проблема в том, что при использовании рекомендательной системы для сбора данных мы получаем смещенное и неполное представление о предпочтениях пользователей: мы видим отклики пользователей только на те продукты, что им были рекомендованы, а все прочие остаются за кадром. Кроме того, в некоторых случаях мы можем не получить вообще никакой информации о пользователях, которым не было дано рекомендаций (например, на аукционе рекламы может случиться, что цена, предложенная за размещение объявления, ниже минимальной цены или не стала победителем, так что объявление не показано вовсе). Важно, что у нас нет информации о том, что случилось бы, если бы были рекомендованы какие-то другие продукты. Тут можно провести аналогию с обучением классификатора, когда для каждого обучающего примера $\boldsymbol{x}$ выбирается один класс $\hat{y}$ (обычно класс с наибольшей вероятностью согласно модели) и в качестве обратной связи мы узнаем только, правильный это класс или нет. Очевидно, что каждый пример несет меньше информации, чем в случае обучения с учителем, когда известна истинная метка y, поэтому необходимо больше примеров. Хуже того, если не проявить осторожность, то можно получить систему, которая будет принимать неверные решения, сколько бы данных ни подать ей на вход, потому у правильного решения изначально была очень низкая вероятность: пока обучаемая система не выберет это правильное решение, она не узнает, что оно правильно. Это похоже на ситуацию в обучении с подкреплением, когда наблюдаемой величиной является только вознаграждение за выбранное действие. В общем случае обучение с подкреплением может содержать последовательность из многих действий и многих вознаграждений. Сценарий с бандитами – это частный случай обучения с подкреплением, когда обучаемый предпринимает единственное действие и получает единственное вознаграждение. Проблема бандита проще в том смысле, что обучаемый знает, какое вознаграждение с каким действием ассоциировано. В общем же случае большое или малое вознаграждение может быть вызвано как недавним действием, так и действием в отдаленном прошлом. Термин «контекстуальные бандиты» относится к случаю, когда действие предпринято в контексте некоторой входной переменной, которая может повлиять на решение. Например, мы знаем как минимум идентификатор пользователя и хотим выбрать для него продукт. Отображение контекста на действие называют также

политикой. Петля обратной связи между обучаемым и распределением данных (которое теперь зависит от действий обучаемого) – центральный вопрос в литературе по обучению с подкреплением и бандитам.

Для обучения с подкреплением требуется выбрать компромисс между **исследованием** (exploration) и **использованием** (exploitation). Под использованием понимается выполнение действий, вытекающих из текущей наилучшей версии обученной политики, – действий, которые, как мы знаем, повлекут за собой большое вознаграждение. А исследование – это выполнение действий, направленных специально на получение дополнительных обучающих данных. Если мы знаем, что при данном контексте $\boldsymbol{x}$ действие a принесет вознаграждение 1, то это еще не значит, что это максимально возможное вознаграждение. Мы можем использовать текущую политику и продолжать выполнять действие a, чтобы более-менее гарантированно получить вознаграждение 1. Но можем и заняться исследованием, попробовав действие a'. Мы не знаем, что произойдет, если выполнить действие a'. Мы надеемся получить вознаграждение 2, но рискуем остаться с вознаграждением 0. Но в любом случае приобретем какие-то знания.

Есть разные способы реализации исследования: можно время от времени предпринимать случайные действия в расчете охватить все пространство возможных действий, а можно положить в основу модель, которая вычисляет действие в зависимости от ожидаемого вознаграждения и заложенной в модель степени неопределенности этого вознаграждения.

Что предпочесть – исследование или использование, зависит от многих факторов. Один из самых важных – интересующий нас временной масштаб. Если у агента имеется ограниченное время для накопления вознаграждения, то мы предпочли бы использование. Если же времени достаточно, то стоит начать с исследования, чтобы планировать будущие действия более эффективно, опираясь на полученные знания. Обучив достаточно хорошую политику, можно переходить к использованию.

В обучении с учителем не нужно выбирать между исследованием и использованием, потому что сигнал от учителя всегда говорит, какой выход правилен для данного входа. Нет нужды пробовать различные выходы, чтобы понять, удастся ли улучшить текущий оптимальный выход модели, – мы заведомо знаем, что лучшим выходом является метка.

Помимо компромисса между исследованием и использованием, в контексте обучения с подкреплением возникает еще одна трудность: оценка и сравнение различных политик. Обучение с подкреплением предполагает взаимодействие между обучаемым и окружением. Эта петля обратной связи означает, что нельзя просто оценить качество обучаемой системы, пользуясь фиксированным тестовым набором входных значений. Сама политика определяет, какие входы будут предъявлены. В работе Dudik et al. (2011) представлены методы оценки контекстуальных бандитов.

12.5.2. Представление знаний, рассуждения и ответы на вопросы

Глубокому обучению удалось добиться больших успехов в языковом моделировании, машинном переводе и обработке естественных языков благодаря использованию погружений для символов (Rumelhart et al., 1986a) и слов (Deerwester et al., 1990; Bengio et al., 2001). Погружения представляют семантические знания об отдельных словах и концепциях. На переднем крае исследований находится разработка погружений

для фраз и связей между словами и фактами. В поисковых системах для этой цели уже используется машинное обучение, но еще многое предстоит сделать для улучшения этих продвинутых представлений.

12.5.2.1. Знания, отношения и ответы на вопросы

Интересное направление исследований – выяснить, как можно обучить распределенные представления улавливать **отношения** между двумя сущностями. Такие связи помогают формализовать факты, касающиеся объектов и их взаимодействий.

В математике **бинарным отношением** называется множество упорядоченных пар объектов. Если пара объектов присутствует в этом множестве, то говорят, что между этими объектами имеется отношение. Например, можно задать отношение «меньше» на множестве сущностей {1, 2, 3}, определив множество упорядоченных пар $\mathbb{S} = \{(1, 2), (1, 3), (2, 3)\}$. После того как отношение определено, его можно использовать как наречие (или глагол – в зависимости от названия отношения). Поскольку $(1, 2) \in \mathbb{S}$, мы говорим, что 1 меньше 2. А так как $(2, 1) \notin \mathbb{S}$, мы не можем сказать, что 2 меньше 1. Конечно, связанные отношением сущности необязательно должны быть числами. Можно было бы определить отношение `имеет_тип`, содержащее кортежи вида (`собака`, `млекопитающее`).

В контексте ИИ мы воспринимаем отношение как предложение в синтаксически простом сильно структурированном языке. Отношение играет роль глагола, а два его аргумента – роли подлежащего (субъекта) и дополнения (объекта). Предложения имеют вид тройки лексем

(подлежащее; глагол; дополнение) (12.21)

со значениями

(сущность_i; отношение_j; сущность_k). (12.22)

Можно определить также **атрибут** – концепцию, аналогичную отношению, с тем отличием, что атрибут принимает всего один аргумент:

(сущность_i; атрибут_j). (12.23)

Например, можно определить атрибут `имеет_мех` и применять его к сущностям типа `собака`.

Во многих приложениях требуется представлять отношения и рассуждать о них. Как лучше всего сделать это в контексте нейронных сетей? Разумеется, моделям машинного обучения нужны обучающие данные. Мы можем вывести отношения между сущностями из обучающих наборов данных, содержащих неструктурированные предложения на естественном языке. Но есть и структурированные базы данных, в которых отношения определены явно. Как правило, для этой цели используются **реляционные базы данных**, в которых хранится информация того же сорта, правда, не представленная в виде троек лексем. Если база данных предназначена для передачи системе искусственного интеллекта общеизвестных знаний о повседневной жизни или специальных знаний о предметной области, то мы называем ее **базой знаний**. Существуют как общие базы знаний: `Freebase`, `OpenCyc`, `WordNet`, `Wikibase`[1], так и специа-

[1] Размещенные на сайтах freebase.com, cyc.com/opencyc, wordnet.princeton.edu, wikiba.se соответственно.

лизированные, например GeneOntology[1]. Представления сущностей и отношений можно обучить, рассматривая каждую тройку в базе знаний как обучающий пример и максимизируя целевую функцию, которая улавливает их совместное распределение (Bordes et al., 2013a).

Помимо обучающих данных, нам нужно также определить семейство подлежащих обучению моделей. Общий подход – распространить нейронные языковые модели на моделирование сущностей и отношений. Результатом обучения нейронной языковой модели является вектор, который дает распределенное представление каждого слова. Она также обучается взаимодействиям между словами, например: какое слово может встретиться после заданной последовательности слов, для чего обучаются функции этих векторов. Этот подход можно распространить на сущности и отношения, если обучить вектор погружений для каждого отношения. На самом деле параллель между моделированием языка и моделированием знаний, закодированных в виде отношений, настолько тесная, что исследователи обучали представления таких сущностей, используя одновременно базы знаний и предложения на естественном языке (Bordes et al., 2011, 2012; Wang et al., 2014a) или комбинируя данные из нескольких реляционных баз данных (Bordes et al., 2013b). Есть много вариантов параметризации такой модели. В ранних работах по обучению отношений между сущностями (Paccanaro and Hinton, 2000) были предложены параметрические формы с сильными ограничениями («линейные реляционные погружения»), причем для представления отношений и сущностей часто использовались различные формы. Например, в работах Paccanaro and Hinton (2000) и Bordes et al. (2011) использовались векторы для сущностей и матрицы для отношений, обосновывая это тем, что отношение выступает в роли оператора над сущностями. Можно также рассматривать отношения как любую другую сущность (Bordes et al., 2012), что позволяет высказывать утверждения относительно отношений, но большей гибкостью обладает механизм, который комбинирует их с целью моделирования совместного распределения.

В качестве примера практического применения таких моделей можно назвать предсказание связей – отсутствующих ребер в графе знаний. Это форма обобщения на новые факты на основе старых фактов. Большинство существующих ныне баз знаний было построено вручную, поэтому многие, а быть может, и большинство истинных отношений в базе отсутствует. Примеры таких приложений приведены в работах Wang et al. (2014b), Lin et al. (2015) и Garcia-Duran et al. (2015).

Оценка качества модели для задачи предсказания связей вызывает трудности, потому что у нас имеется только набор положительных примеров (заведомо истинных фактов). Если модель предлагает факт, отсутствующий в наборе, мы не знаем наверняка, то ли модель допустила ошибку, то ли действительно открыла новый, ранее неизвестный факт. Поэтому все метрики не вполне точны и основаны на проверке того, как модель ранжирует зарезервированный набор заведомо правильных фактов по сравнению с другими фактами, которые могут быть и неправильны. Стандартный способ конструирования интересных, предположительно отрицательных примеров (фактов, которые с большой вероятностью ложны) – начать с истинного факта и создать его искаженные версии, например подменив одну сущность в отношении другой, выбранной случайным образом. Популярная метрика «точность на 10 процентах»

[1] geneontology.org.

подсчитывает, сколько раз модель включает «правильный» факт в первые 10% всех искаженных версий этого факта.

Еще одно применение баз знаний и их распределенных представлений – **разрешение неоднозначности смысла слов** (Navigli and Velardi, 2005; Bordes et al., 2012), т. е. решение вопроса о том, какое значение слова соответствует заданному контексту.

В конечном итоге знание отношений в сочетании с процессом рассуждения и пониманием естественного языка позволят нам построить общую вопросно-ответную систему. Такая система должна обрабатывать входную информацию и запоминать важные факты, организуя их таким образом, чтобы впоследствии их можно было извлечь и рассуждать о них. Это остается трудной проблемой, решенной только в ограниченном «игрушечном» окружении. В настоящее время наилучшим подходом к запоминанию и извлечению конкретных декларативных фактов считается использование механизма явной памяти, описанного в разделе 10.2. Сети с памятью также впервые были использованы для создания игрушечной вопросно-ответной системы (Weston et al., 2014). В работе Kumar et al. (2015) предложено обобщение, в котором рекуррентные сети GRU применяются для чтения входных данных в памяти и порождения ответа при заданном состоянии памяти.

Глубокое обучение применялось и ко многим другим задачам, помимо описанных выше, и без сомнения найдет еще больше применений уже после выхода книги из печати. Невозможно представить себе нечто, хотя бы отдаленно напоминающее исчерпывающий трактат на эту тему. Приведенный обзор – репрезентативная выборка из того, что было возможно на момент написания книги.

На этом заканчивается часть II, в которой описаны современные практические применения глубоких сетей, охватывающие все наиболее успешные методы. Вообще говоря, в этих методах используется градиент функции стоимости для нахождения таких параметров модели, которые лучше всего аппроксимируют некоторую желательную функцию. При наличии достаточного объема обучающих данных это чрезвычайно мощный подход. Далее мы переходим к части III, где ступаем на территорию активных исследований – методов, которые должны работать при меньшем объеме обучающих данных или решать более разнообразные задачи. Проблемы здесь труднее и не так близки к решению, как в описанных выше ситуациях.

Часть III

ИССЛЕДОВАНИЯ ПО ГЛУБОКОМУ ОБУЧЕНИЮ

В этой части книги описываются передовые подходы к глубокому обучению, развиваемые в настоящее время исследовательским сообществом.

В предыдущих частях мы показали, как решаются задачи обучения с учителем: как обучить отображение одного вектора на другой, если имеется достаточно примеров такого отображения.

Но не все интересующие нас задачи попадают в эту категорию. Иногда требуется генерировать новые примеры, определить вероятность какой-то точки в пространстве событий, обработать отсутствие значений или воспользоваться большим набором непомеченных примеров либо примерами для родственных задач. У современных промышленных приложений есть существенный недостаток: для достижения приемлемой верности алгоритмам обучения нужно много подготовленных учителем данных. В этой части книги мы обсудим некоторые теоретически мыслимые подходы к уменьшению объема размеченных данных, необходимого для качественной работы существующих моделей и их применения к более широкому кругу задач. Обычно для достижения этих целей приходится прибегать к той или иной форме обучения без учителя или с частичным привлечением учителя. Придумано много алгоритмов глубокого обучения без учителя, но ни один из них не способен продемонстрировать такое же качество, как при решении задач обучения с учителем. Мы опишем существующие подходы к обучению без учителя и некоторые популярные идеи о том, как добиться прогресса в этой области.

Основная причина трудностей обучения без учителя – высокая размерность моделируемых случайных величин. Из-за этого возникают две проблемы: статистическая и вычислительная. *Статистическая проблема* касается обобщения: количество конфигураций, которые мы хотели бы различать, может экспоненциально возрастать с ростом числа представляющих интерес измерений и в итоге очень быстро оказывается намного больше, чем количество доступных примеров (или превышает возможности ограниченных вычислительных ресурсов). *Вычислительная проблема* многомерных распределений связана с тем, что многие алгоритмы обучения или использования уже обученной модели (особенно те, что основаны на оценивании явной функции вероятности) подразумевают вычисления, объем которых экспоненциально зависит от числа измерений и превосходит возможности оборудования.

В вероятностных моделях вычислительные проблемы возникают из-за необходимости произвести неразрешимый вывод или нормировать распределение.

- *Неразрешимый вывод.* Вывод обсуждается преимущественно в главе 19. Речь идет об угадывании вероятных значений некоторых величин a при условии других величин b в модели, которая хранит совместное распределение величин a, b и c. Чтобы просто вычислить такие условные вероятности, необходимо просуммировать по значениям величин c, а также вычислить нормировочную постоянную, что требует суммирования по значениям a и c.
- *Неразрешимые нормировочные постоянные (статистическая сумма).* Статистическая сумма обсуждается в основном в главе 18. Нормировочные постоянные функций вероятности возникают как при выводе (см. выше), так и при обучении. Такие постоянные входят во многие вероятностные модели. К сожалению, для обучения такой модели часто необходимо вычислять градиент логарифма статистической суммы по параметрам модели. Эти вычисления в общем случае так же сложны, как и вычисление самой статистической суммы. Для работы со статистической суммой (вычисления ее самой или ее градиента) часто применяются методы Монте-Карло по схеме марковских цепей (Monte Carlo Markov chain – MCMC) (глава 17). Увы, они испытывают трудности, когда у распределения много мод и они далеко отстоят друг от друга, что особенно характерно для пространств высокой размерности (раздел 17.5).

Один из способов борьбы с такими неразрешимыми вычислениями – аппроксимация, и в этом направлении предложено много методов, обсуждаемых ниже. Другой интересный подход – вообще избежать таких вычислений, и методы, в которых удается обойтись без них, выглядят очень соблазнительно. С этой целью в последние годы предложено несколько порождающих моделей. Широкий спектр современных подходов к порождающему моделированию обсуждается в главе 20.

Часть III представляет особый интерес для исследователей – тех, кто хочет понять всю широту направлений, изучаемых в области глубокого обучения, и продвинуться на пути к настоящему искусственному интеллекту.

Глава 13

Линейные факторные модели

На передовых рубежах глубокого обучения часто требуется построить вероятностную модель входа, $p_{\text{model}}(\boldsymbol{x})$. В принципе, в такой модели может использоваться вероятностный вывод для предсказания любой переменной в ее окружении при известных других переменных. Во многих моделях имеются также латентные переменные $\boldsymbol{h}$, так что $p_{\text{model}}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{h}} p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$. Латентные переменные дают еще один способ представления данных. Распределенные представления, основанные на латентных переменных, могут пользоваться всеми преимуществами обучения представлений, о которых мы говорили при обсуждении глубоких сетей прямого распространения и рекуррентных сетей.

В этой главе мы опишем простейшие вероятностные модели с латентными переменными: линейные факторные модели. Иногда они используются в качестве составных частей смесовых моделей (Hinton et al., 1995a; Ghahramani and Hinton, 1996; Roweis et al., 2002) или более крупных глубоких вероятностных моделей (Tang et al., 2012). На их примере можно также продемонстрировать основные подходы к построению порождающих моделей, которые обобщаются в более передовых глубоких моделях.

Линейная факторная модель определяется стохастической линейной функцией декодера, которая генерирует $\boldsymbol{x}$ посредством прибавления шума к результату линейного преобразования $\boldsymbol{h}$.

Эти модели интересны тем, что позволяют выявить объясняющие факторы, имеющие простое совместное распределение. Благодаря простоте линейных декодеров эти модели стали первыми моделями с латентными переменными, подвергшимися пристальному изучению.

Линейная факторная модель описывает порождающий данные процесс следующим образом. Сначала производится выборка объясняющих факторов $\boldsymbol{h}$ из распределения:

$$\mathbf{h} \sim p(\boldsymbol{h}), \tag{13.1}$$

где $p(\boldsymbol{h})$ – факторное распределение: $p(\boldsymbol{h}) = \prod_i p(h_i)$, из которого легко производить выборку. Затем мы производим выборку вещественных наблюдаемых переменных при условии факторов

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise}, \tag{13.2}$$

где шум noise обычно имеет нормальное диагональное (независимое по разным осям) распределение. Это показано на рис. 13.1.

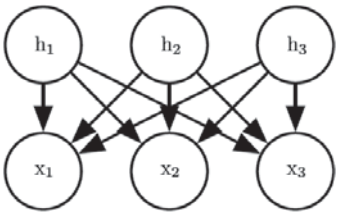

Рис. 13.1 ❖ Ориентированная графическая модель, описывающая семейство линейных факторных моделей. Предполагается, что наблюдаемый вектор $\boldsymbol{x}$ является линейной комбинацией независимых латентных факторов $\boldsymbol{h}$ и шума. В различных моделях, например в вероятностном методе главных компонент, факторном анализе и анализе независимых компонент (ICA), делаются различные предположения о форме шума и априорном распределении $p(\boldsymbol{h})$

13.1. Вероятностный PCA и факторный анализ

В **факторном анализе** (Bartholomew, 1987; Basilevsky, 1994) латентная переменная имеет априорное нормальное распределение с единичной дисперсией

$$\mathbf{h} \sim \mathcal{N}(\boldsymbol{h}; \mathbf{0}, \boldsymbol{I}), \tag{13.3}$$

а наблюдаемые переменные x_i предполагаются **условно независимыми** при условии $\boldsymbol{h}$. Точнее говоря, предполагается, что шум выбирается из нормального распределения с диагональной ковариационной матрицей $\psi = \text{diag}(\boldsymbol{\sigma}^2)$, где $\boldsymbol{\sigma}^2 = [\sigma_1^2, \sigma_2^2, ..., \sigma_n^2]^\top$ – вектор дисперсий отдельных переменных.

Таким образом, роль латентных переменных – в *улавливании зависимостей* между различными наблюдаемыми переменными x_i. Действительно, легко показать, что $\boldsymbol{x}$ – просто случайная величина с многомерным нормальным распределением:

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \psi). \tag{13.4}$$

Чтобы переформулировать метод главных компонент (PCA) в вероятностном контексте, мы можем внести небольшую модификацию в модель факторного анализа, сделав условные дисперсии σ_i^2 равными. В таком случае ковариация $\boldsymbol{x}$ равна просто $\boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}$, где σ^2 – теперь скаляр. Тогда условное распределение имеет вид

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}), \tag{13.5}$$

или, эквивалентно,

$$\mathbf{x} = \boldsymbol{W}\mathbf{h} + \boldsymbol{b} + \sigma\mathbf{z}, \tag{13.6}$$

где $\mathbf{z} \sim \mathcal{N}(\boldsymbol{z}; \mathbf{0}, \boldsymbol{I})$ – гауссов шум. Далее, как показано в работе Tipping and Bishop (1999), можно использовать итеративный EM-алгоритм для оценивания параметров $\boldsymbol{W}$ и σ^2.

В этой модели **вероятностного** PCA предполагается, что своей вариативностью данные обязаны прежде всего латентным переменным $\boldsymbol{h}$ с точностью до небольшой остаточной **ошибки реконструкции** σ^2. Как показано в работе Tipping and Bishop (1999), вероятностный PCA превращается в PCA, когда $\sigma \to 0$. В этом случае услов-

ное математическое ожидание $\boldsymbol{h}$ при условии $\boldsymbol{x}$ становится ортогональной проекцией $\boldsymbol{x} - \boldsymbol{b}$ на пространство, натянутое на d столбцов $\boldsymbol{W}$, как в PCA.

При $\sigma \to 0$ модель плотности, определяемая вероятностным PCA, становится очень острой в направлении тех d измерений, которые натянуты на столбцы $\boldsymbol{W}$. Поэтому модель может назначать очень низкое правдоподобие данным, если они в действительности не образуют кластера в окрестности этой гиперплоскости.

13.2. Анализ независимых компонент (ICA)

Анализ независимых компонент (independent component analysis – ICA) – один из самых старых алгоритмов обучения представлений (Herault and Ans, 1984; Jutten and Herault, 1991; Comon, 1994; Hyvärinen, 1999; Hyvärinen et al., 2001a; Hinton et al., 2001; Teh et al., 2003). Идея этого подхода к моделированию линейных факторов заключается в том, чтобы представить наблюдаемый сигнал в виде линейной комбинации нескольких составляющих. Предполагается, что эти сигналы полностью независимы, а не просто не коррелированы[1].

Под общим названием ICA объединено несколько разных методологий. На другие описанные здесь порождающие модели больше всего похож вариант (Pham et al., 1992), в котором обучается полностью параметрическая порождающая модель. Априорное распределение объясняющих факторов $p(\boldsymbol{h})$ должно быть зафиксировано пользователем заранее. Затем модель детерминированно порождает $\boldsymbol{x} = \boldsymbol{Wh}$. Для определения $p(\boldsymbol{x})$ мы можем произвести нелинейную замену переменных (пользуясь формулой 3.47). Далее обучение модели протекает как обычно, с применением максимального правдоподобия.

Обоснование этого подхода состоит в том, что, выбирая $p(\boldsymbol{h})$ независимым, мы можем восстановить объясняющие факторы, которые настолько близки к независимым, насколько возможно. Обычно это используется не для того, чтобы уловить высокоуровневые абстрактные каузальные факторы, а чтобы восстановить низкоуровневые сигналы, которые можно смешать. При такой постановке каждый обучающий пример – это один момент времени, каждый x_i – одно наблюдение смешанных сигналов, полученное от датчика, а каждый h_i – одна оценка одного из исходных сигналов. Например, пусть одновременно разговаривают n человек. Если в разных точках установлено n микрофонов, то метод ICA позволяет обнаружить изменения громкости каждого говорящего, зафиксированной каждым микрофоном, и разделить сигналы таким образом, что каждая компонента h_i содержит отчетливую речь одного человека. Это часто применяется для снятия электроэнцефалограмм – электрических сигналов мозга. На голове пациента закрепляется несколько электродов для измерения электрических сигналов тела. Обычно экспериментатора интересуют только сигналы мозга, но сигналы, исходящие от сердца и глаз, достаточно сильны, чтобы исказить результаты измерений, произведенных на черепе. Электроды снимают смешанный сигнал, поэтому для отделения электрической сигнатуры сердца от сигналов мозга, а также для разделения сигналов от различных участков мозга необходимо применение ICA.

Как уже было сказано, существует много вариантов ICA. Некоторые добавляют шум в процесс генерации $\boldsymbol{x}$ вместо использования детерминированного декодера.

---

[1] Обсуждение вопроса о том, чем некоррелированные величины отличаются от независимых, см. в разделе 3.8.

В большинстве не используется критерий максимального правдоподобия, а вместо этого ставится цель сделать элементы $\boldsymbol{h} = \boldsymbol{W}^{-1}\boldsymbol{x}$ независимыми друг от друга. Есть много критериев, позволяющих добиться этой цели. В формуле (3.47) необходимо вычислять определитель $\boldsymbol{W}$, а это дорогая и вычислительно неустойчивая операция. В некоторых вариантах ICA эту проблему обходят, требуя, чтобы матрица $\boldsymbol{W}$ была ортогональной.

Во всех вариантах ICA требуется, чтобы распределение $p(\boldsymbol{h})$ было негауссовым. Связано это с тем, что если $p(\boldsymbol{h})$ – независимое априорное распределение с гауссовыми компонентами, то матрица $\boldsymbol{W}$ определена неодозначно. Одно и то же распределение $p(\boldsymbol{x})$ можно получить при разных значениях $\boldsymbol{W}$. В этом состоит разительное отличие от других линейных факторных моделей, в т. ч. вероятностного PCA и факторного анализа, где часто требуется, чтобы $p(\boldsymbol{h})$ было гауссовым, поскольку тогда многие операции с моделью могут быть выражены в замкнутой форме. В подходе на основе максимального правдоподобия, когда пользователь задает распределение явно, типичным выбором является $p(h_i) = (d/dh_i)\sigma(h_i)$. Эти негауссовы распределения обычно выбирают так, чтобы пик в окрестности нуля был выше, чем у гауссова, поэтому в большинстве реализаций ICA обучаются разреженные признаки.

Многие варианты ICA не являются порождающими моделями в обычном смысле слова. В этой книге порождающая модель либо представляет распределение $p(\boldsymbol{x})$, либо умеет производить выборку из него. Многие варианты ICA знают только, как выполнить преобразование между $\boldsymbol{x}$ и $\boldsymbol{h}$, но не имеют средств для представления $p(\boldsymbol{h})$, а потому не подразумевают никакого конкретного распределения $p(\boldsymbol{x})$. Так, во многих вариантах ICA ставится цель увеличить куртозис $\boldsymbol{h} = \boldsymbol{W}^{-1}\boldsymbol{x}$, поскольку высокий куртозис является признаком негауссовости $p(\boldsymbol{h})$, но этой цели можно достичь и без явного представления $p(\boldsymbol{h})$. Дело в том, что ICA чаще используется как аналитический инструмент для разделения сигналов, а не для порождения данных или оценки плотности.

Как PCA можно обобщить на нелинейные автокодировщики, описанные в главе 14, так и ICA обобщается на нелинейные порождающие модели, в которых для генерации наблюдаемых данных используется нелинейная функция f. Первой работой по нелинейному ICA была статья Hyvärinen and Pajunen (1999), а его успешное применение к ансамблевому обучению описано в работах Roberts and Everson (2001) и Lappalainen et al. (2000). Еще одно нелинейное обобщение ICA – **нелинейное оценивание независимых компонент** (nonlinear independent components estimation – NICE) (Dinh et al., 2014), когда строится последовательность обратимых преобразований (ступеней кодировщика), обладающая тем свойством, что определитель матрицы Якоби каждого преобразования допускает эффективное вычисление. Это позволяет точно вычислить правдоподобие; как и ICA, NICE пытается преобразовать данные в пространство, где они имеют факторизованное маргинальное распределение, но теперь, благодаря нелинейности кодировщика, шансов на успех этого предприятия больше. Поскольку с кодировщиком ассоциирован декодер, являющийся его точным обращением, то выборка из модели не представляет никаких трудностей (нужно сначала произвести выборку из $p(\boldsymbol{h})$, а затем применить декодер).

Еще одно обобщение ICA – обучение групп признаков, когда статистическая зависимость допускается внутри группы, но запрещается между группами (Hyvärinen and Hoyer, 1999; Hyvärinen et al., 2001b). Если выбираются непересекающиеся груп-

пы взаимосвязанных блоков, то метод называется **анализом независимых подпространств** (independent subspace analysis). Можно также назначить пространственные координаты каждому скрытому блоку и сформировать пересекающиеся группы пространственно соседних блоков. Это побуждает близкие блоки обучаться похожим признакам. В применении к естественным изображениям этот **топографический вариант ICA** обучает фильтры Габора, так чтобы у соседних признаков была одинаковая ориентация, местоположение или частота. Внутри каждой области встречается много разных сдвигов фаз похожих функций Габора, поэтому пулинг по небольшим областям дает инвариантность относительно параллельного переноса.

13.3. Анализ медленных признаков

Анализ медленных признаков (slow feature analysis – SFA) – линейная факторная модель, в которой для обучения инвариантных признаков используется информация от сигналов времени (Wiskott and Sejnowski, 2002).

В основе анализа медленных признаков лежит общий **принцип медленности**. Идея в том, что важные характеристики сцены изменяются очень медленно, по сравнению с отдельными измерениями, из которых складывается описание сцены. Например, в компьютерном зрении значения отдельных пикселей могут изменяться очень быстро. Если имеется ряд изображений зебры, бегущей слева направо, то значения пикселей быстро меняются с черного на белый и наоборот. Однако признак, показывающий, присутствует в изображении зебра или нет, не изменяется вовсе, а признак, описывающий положение зебры, изменяется медленно. Поэтому возникает желание регуляризировать модель, так чтобы она обучилась признакам, которые медленно изменяются во времени.

Принцип медленности появился намного раньше анализа медленных признаков и с успехом применялся к широкому кругу моделей (Hinton, 1989; Földiák, 1989; Mobahi et al., 2009; Bergstra and Bengio, 2009). В общем случае принцип медленности можно применить к любой дифференцируемой модели, обученной методом градиентного спуска. Для этого нужно включить в функцию стоимости член вида

$$\lambda \sum_t L(f(\boldsymbol{x}^{(t+1)}), f(\boldsymbol{x}^{(t)})), \tag{13.5}$$

где λ – гиперпараметр, определяющий силу регуляризирующего члена, t – индекс временной последовательности примеров, f – экстрактор подлежащих регуляризации признаков, а L – функция потерь, измеряющая расстояние между $f(\boldsymbol{x}^{(t)})$ и $f(\boldsymbol{x}^{(t+1)})$. В качестве L обычно берут квадрат среднеквадратического отклонения.

Анализ медленных признаков – особенно эффективное применение принципа медленности. Эффективность объясняется тем, что модель применяется к экстрактору линейных признаков и потому может быть обучена в замкнутой форме. Как и некоторые варианты ICA, SFA, строго говоря, не является порождающей моделью, поскольку определяет линейное отображение между пространством входов и пространством признаков, но не определяет априорного распределения в пространстве признаков и потому не налагает никакого распределения $p(\boldsymbol{x})$ на пространство входов.

Алгоритм SFA (Wiskott and Sejnowski, 2002) требует, чтобы $f(\boldsymbol{x}; \boldsymbol{\theta})$ было линейным преобразованием и решает задачу оптимизации

$$\min_{\boldsymbol{\theta}} \mathbb{E}_t (f(\boldsymbol{x}^{(t+1)})_i - f(\boldsymbol{x}^{(t)})_i)^2 \tag{13.6}$$

при ограничениях

$$\mathbb{E}_t f(\boldsymbol{x}^{(t)})_i = 0 \tag{13.9}$$

и

$$\mathbb{E}_t[f(\boldsymbol{x}^{(t)})_i^2] = 1. \tag{13.10}$$

Среднее значение обученного признака должно быть равно нулю, чтобы у задачи было единственное решение; в противном случае можно было бы прибавить ко всем признакам одну и ту же константу и получить другое решение с тем же значением целевой функции медленности. А дисперсия должна быть равна 1, чтобы предотвратить патологическую ситуацию, когда все признаки оказываются равны 0. Как и в случае PCA, признаки SFA упорядочены, причем на первом месте находится самый медленный. Для обучения нескольким признакам нужно добавить еще ограничение

$$\forall i < j,\ \mathbb{E}_t[f(\boldsymbol{x}^{(t)})_i f(\boldsymbol{x}^{(t)})_j] = 0. \tag{13.11}$$

Это означает отсутствие линейной корреляции между обученными признаками. Не будь этого ограничения, все обученные признаки просто запомнили бы самый медленный сигнал. Можно представить себе и другие механизмы принудительной диверсификации признаков, например минимизацию ошибки реконструкции, но требование отсутствия корреляции допускает простое решение в силу линейности признаков SFA. Задачу оптимизации SFA можно решить в замкнутой форме, применяя методы линейной алгебры.

Обычно SFA применяется для обучения нелинейным признакам посредством применения к $\boldsymbol{x}$ нелинейного расширения базиса до выполнения SFA. Например, часто заменяют $\boldsymbol{x}$ квадратичным расширением базиса – вектором, содержащим элементы $x_i x_j$ для всех i и j. Затем можно составить композицию модулей линейного SFA для обучения глубоких нелинейных экстракторов медленных признаков, для чего нужно несколько раз повторить следующие действия: обучить линейный экстрактор, применить к его выходу нелинейное расширение базиса и обучить еще один линейный экстрактор признаков SFA на результатах расширения.

При обучении на небольших пространственных участках видео естественных сцен SFA с квадратичным расширением базиса обучается признакам, имеющим много общих характеристик со сложными клетками в зоне V1 первичной зрительной коры (Berkes and Wiskott, 2005). При обучении на видео случайного движения в отрисованной компьютером трехмерной сцене глубокая модель SFA обучается признакам, имеющим много общих характеристик с нейронами в мозгу крыс, отвечающими за ориентацию в пространстве (Franzius et al., 2007). Таким образом, модель SFA представляется биологически правдоподобной.

Главное достоинство SFA состоит в том, что можно теоретически предсказать, каким именно признакам обучится SFA, даже в глубокой нелинейной конфигурации. Для этого необходимо знать динамику окружения в терминах конфигурационного пространства (например, для случайного движения в трехмерном отрисованном окружении теоретический анализ основывается на знании распределения вероятности положения и скорости перемещения камеры). Зная, как на самом деле изменяются объясняющие факторы, можно аналитически найти оптимальные функции, выра-

жающие эти факторы. На практике эксперименты с применением глубокой модели SFA к имитированным данным, похоже, восстанавливают теоретически предсказанные функции. Это большое преимущество, по сравнению с другими алгоритмами обучения, где функция стоимости сильно зависит от конкретных значений пикселей, из-за чего установить, каким функциям обучится модель, гораздо труднее.

Глубокий SFA применялся также для обучения признакам для распознавания объектов и оценки расположения (Franzius et al., 2008). Пока еще принцип медленности не лег в основу современных передовых приложений. Не ясно, что именно ограничивает его качество. Мы подозреваем, что априорная гипотеза медленности слишком сильная и что лучше бы не требовать, чтобы признаки были приблизительно постоянны, а наложить ограничение, что их должно быть легко предсказать при переходе от предыдущего шага к следующему. Положение объекта – полезный признак вне зависимости от того, велика или мала его скорость, но принцип медленности поощряет модель игнорировать положение быстро перемещающихся объектов.

13.4. Разреженное кодирование

Разреженное кодирование (Olshausen and Field, 1996) – это линейная факторная модель, которая активно изучалась как механизм обучения признакам без учителя и выделения признаков. Строго говоря, термин «разреженное кодирование» относится к процессу вывода значения $\boldsymbol{h}$ в этой модели, а «разреженное моделирование» – к процессу проектирования и обучения модели, но часто под «разреженным кодированием» понимают то и другое.

Как и в большинстве линейных факторных моделей, здесь используется линейный декодер плюс шум для получения реконструкций $\boldsymbol{x}$ в соответствии с формулой (13.2). Точнее, в моделях разреженного кодирования обычно предполагается, что в линейных факторах присутствует гауссов шум с изотропной точностью β:

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}, \frac{1}{\beta}\boldsymbol{I}). \tag{13.12}$$

Распределение $p(\boldsymbol{h})$ выбирается так, чтобы были острые пики вблизи 0 (Olshausen and Field, 1996). Обычно используются факторизованные распределения Лапласа, Коши или t-распределения Стьюдента. Например, априорное распределение Лапласа, параметризованное штрафом разреженности λ, имеет вид:

$$p(h_i) = \text{Laplace}\left(h_i; 0, \frac{2}{\lambda}\right) = \frac{\lambda}{4} e^{-\frac{1}{2}\lambda|h_i|}, \tag{13.13}$$

а t-распределение Стьюдента:

$$p(h_i) \propto \frac{1}{\left(1 + \frac{h_i^2}{v}\right)^{\frac{v+1}{2}}}. \tag{13.14}$$

Обучить модель разреженного кодирования методом максимального правдоподобия невозможно из-за гигантского объема вычислений. Вместо этого чередуют кодирование данных с обучением декодера для лучшей реконструкции данных при задан-

ной кодировке. В разделе 19.3 мы приведем теоретическое обоснование этого подхода в качестве аппроксимации максимального правдоподобия.

Для моделей типа PCA мы уже видели, как используется параметрическая функция кодирования, которая предсказывает $\boldsymbol{h}$ и включает только умножение на матрицу весов. В разреженном кодировании используется не параметрический кодировщик, а алгоритм оптимизации, который решает задачу нахождения одного наиболее вероятного значения кода:

$$\boldsymbol{h}^* = f(\boldsymbol{x}) = \arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{x}). \tag{13.15}$$

В сочетании с формулами (13.13) и (13.12) получаем следующую задачу оптимизации:

$$\arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{x}) \tag{13.16}$$

$$= \arg\max_{\boldsymbol{h}} \log p(\boldsymbol{h} \mid \boldsymbol{x}) \tag{13.17}$$

$$= \arg\min_{\boldsymbol{h}} \lambda \|\boldsymbol{h}\|_1 + \beta \|\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}\|_2^2, \tag{13.18}$$

где мы опустили члены, не зависящие от $\boldsymbol{h}$, и для простоты поделили на положительные масштабные коэффициенты.

Поскольку для $\boldsymbol{h}$ используется норма L^1, эта процедура даст разреженный вектор $\boldsymbol{h}^*$ (см. раздел 7.1.2).

Чтобы обучить модель, а не просто произвести вывод, мы чередуем минимизацию по $\boldsymbol{h}$ с минимизацией по $\boldsymbol{W}$. Здесь β рассматривается как гиперпараметр. Обычно ему присваивают значение 1, потому что в этой задаче оптимизации у него общая роль с λ, а заводить два гиперпараметра нет смысла. В принципе, можно было бы рассматривать β как параметр модели и обучить его. В нашем изложении отброшены некоторые члены, не зависящие от $\boldsymbol{h}$, но зависящие от β. Если мы хотим обучить β, то эти члены следует включить, иначе β выродится в 0.

Не во всех подходах к разреженному кодированию явно строятся $p(\boldsymbol{h})$ и $p(\boldsymbol{x} \mid \boldsymbol{h})$. Иногда мы хотим только обучить словарь признаков со значениями активации, которые часто будут равны 0 при выделении с помощью этой процедуры вывода.

Если $\boldsymbol{h}$ выбирается из априорного распределения Лапласа, то обращение любого элемента $\boldsymbol{h}$ в 0 – невозможное событие. Сама порождающая модель не особенно разреженная, таковым является только экстрактор признаков. В работе Goodfellow et al. (2013d) описан приближенный вывод в другом семействе моделей – разреженного кодирования типа Spike-and-Slab, – для которого выборка из априорного распределения обычно содержит нули.

Разреженное кодирование в сочетании с использованием непараметрического кодировщика, в принципе, может минимизировать комбинацию ошибки реконструкции с логарифмической априорной вероятностью лучше, чем любой конкретный параметрической кодировщик. Еще одно преимущество состоит в том, что у кодировщика нет ошибки обобщения. Параметрический кодировщик должен обучиться такому отображению $\boldsymbol{x}$ на $\boldsymbol{h}$, которое хорошо обобщается. Для необычных $\boldsymbol{x}$, не похожих на обучающие данные, обученный параметрический кодировщик может не найти $\boldsymbol{h}$, приводящее к верной реконструкции или к разреженному коду. В подавляющем большинстве формулировок моделей разреженного кодирования, если задача вывода выпуклая,

процедура оптимизации гарантированно находит оптимальный код (за исключением вырожденных случаев, как, например, повторяющиеся векторы весов). Очевидно, что затраты на разреженность и реконструкцию все равно могут быть выше для незнакомых точек, но это из-за ошибки обобщения в весах декодера, а не из-за ошибки обобщения в кодировщике. Благодаря отсутствию ошибки обобщения в процессе разреженного кодирования, основанного на оптимизации, ошибка обобщения в случае использования разреженного кодирования в качестве экстрактора признаков для классификатора может оказаться лучше, чем при использовании параметрической функции для предсказания кода. В работе Coates and Ng (2011) показано, что признаки, найденные с помощью разреженного кодирования, лучше обобщаются в задачах распознавания объектов, чем признаки, найденные родственной моделью, основанной на параметрическом кодировщике, – линейно-сигмоидным автокодировщиком. По следам этой работы в работе Goodfellow et al. (2013d) показано, что вариант разреженного кодирования обобщается лучше, чем другие экстракторы признаков, в режиме, когда доступно очень мало меток (двадцать или даже меньше на каждый класс).

Основной недостаток непараметрического кодировщика в том, что ему требуется больше времени для вычисления $\boldsymbol{h}$ при известном $\boldsymbol{x}$ из-за необходимости выполнять итеративный алгоритм. В параметрическом автокодировщике, который будет описан в главе 14, используется только фиксированное число слоев, часто всего один. Еще один недостаток – сложность обратного распространения через непараметрический кодировщик, что затрудняет предобучение модели разреженного кодирования с помощью критерия без учителя и последующую настройку с помощью критерия с учителем. Существуют модифицированные варианты разреженного кодирования, допускающие приближенные производные, но они не получили широкого распространения (Bagnell and Bradley, 2009).

Разреженное кодирование, как и другие линейные факторные модели, часто порождает плохие примеры, как показано на рис. 13.2. Так бывает, даже когда модель способна хорошо реконструировать данные и предоставляет полезные признаки классификатору. Причина в том, что каждый отдельный признак может быть обучен хорошо, но факторное априорное распределение скрытого кода приводит к тому, что модель включает случайные подмножества всех признаков в каждый сгенерированный пример. Это является стимулом для разработки более глубоких моделей, которые могут применять нефакторное распределение в самом глубоком слое кода, а также для разработки более изощренных мелких моделей.

13.5. Интерпретация PCA в терминах многообразий

Линейные факторные модели, в т. ч. PCA и факторный анализ, можно интерпретировать как обучение многообразия (Hinton et al., 1997). Мы можем рассматривать вероятностный PCA как определение блинообразной области высокой вероятности – нормальное распределение, очень узкое вдоль некоторых осей, как блин вдоль своей вертикальной оси, и сильно вытянутое вдоль остальных осей, как блин вдоль горизонтальных осей (см. рис. 13.3). PCA можно интерпретировать как совмещение этого блина с линейным многообразием в пространстве более высокой размерности. Такая интерпретация относится не только к традиционному PCA, но и к любому линейному автокодировщику, который обучается матрицам $\boldsymbol{W}$ и $\boldsymbol{V}$, добиваясь, чтобы реконструкция $\boldsymbol{x}$ находилась как можно ближе к $\boldsymbol{x}$.

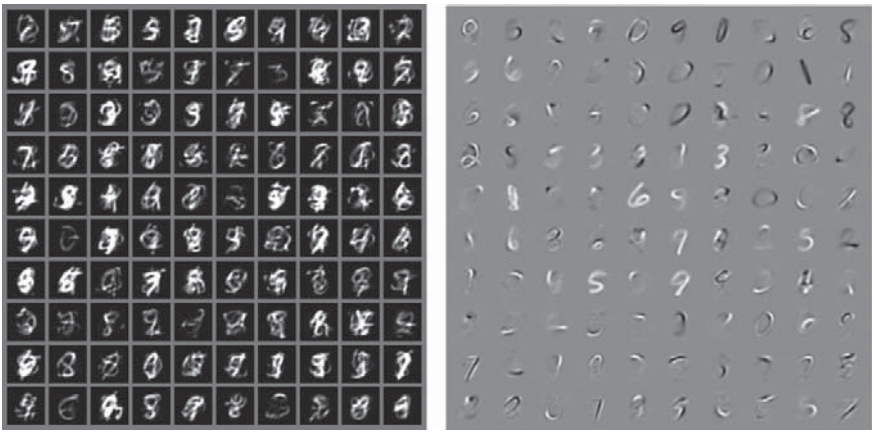

Рис. 13.2 ❖ Образцы примеров и весов из модели разреженного кодирования типа Spike-and-Slab, обученной на наборе данных MNIST. (*Слева*) Выборка из модели не похожа на обучающие примеры. На первый взгляд можно предположить, что модель плохо обучена. (*Справа*). Векторы весов модели обучены представлять росчерки пера и иногда целые цифры. Следовательно, модель обучилась полезным признакам. Проблема в том, что факторное априорное распределение признаков приводит к комбинированию случайных подмножеств признаков. Немногие из таких подмножеств годятся для образования распознаваемой цифры из набора MNIST. Отсюда необходимость разработки порождающих моделей с более мощными распределениями латентных кодов. Рисунок взят из работы Goodfellow et al. (2013d) с разрешения авторов

Пусть кодировщик имеет вид

$$\boldsymbol{h} = f(\boldsymbol{x}) = \boldsymbol{W}^{\top}(\boldsymbol{x} - \boldsymbol{\mu}). \tag{13.19}$$

Кодировщик вычисляет представление $\boldsymbol{h}$ низкой размерности. С точки зрения автокодировщика, мы имеем декодер, вычисляющий реконструкцию

$$\hat{\boldsymbol{x}} = g(\boldsymbol{h}) = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}. \tag{13.20}$$

Выбору линейного кодировщика и декодера, минимизирующих ошибку реконструкции

$$\mathbb{E}[\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2], \tag{13.21}$$

соответствует $\boldsymbol{V} = \boldsymbol{W}, \boldsymbol{\mu} = \boldsymbol{b} = \mathbb{E}[\boldsymbol{x}]$, а столбцы матрицы $\boldsymbol{W}$ образуют ортонормированный базис, определяющий то же подпространство, что и главные собственные векторы ковариационной матрицы

$$\boldsymbol{C} = \mathbb{E}[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^{\top}]. \tag{13.22}$$

В случае PCA столбцами $\boldsymbol{W}$ являются эти собственные векторы, упорядоченные по абсолютной величине соответствующих собственных значений (которые все вещественные и неотрицательные).

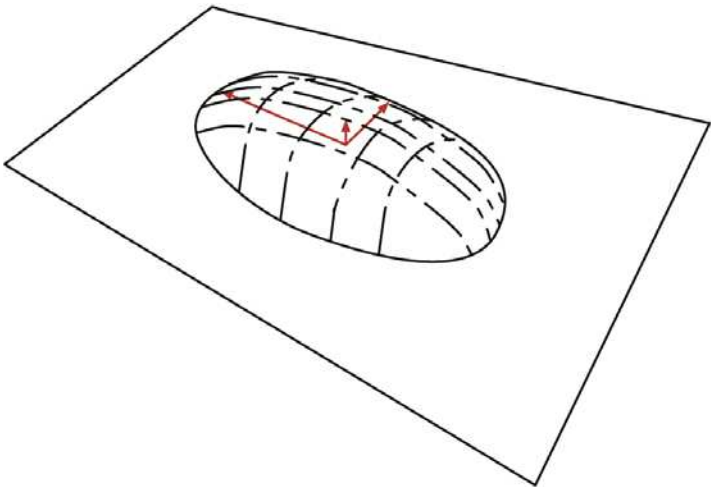

Рис. 13.3 ❖ Плоское нормальное распределение с высокой концентрацией вероятности в окрестности многообразия низкой размерности. На рисунке показана верхняя половина «блина» над «плоскостью многообразия», проходящей через его середину. Дисперсия в направлении, ортогональном многообразию, очень мала (стрелка в направлении наружу от плоскости) и может считаться «шумом», тогда как дисперсии в других направлениях (стрелки внутри плоскости) велики и соответствуют «сигналу» и определяют систему координат для данных пониженной размерности

Можно также показать, что собственное значение λ_i матрицы $\boldsymbol{C}$ соответствует дисперсии $\boldsymbol{x}$ в направлении собственного вектора $\boldsymbol{v}^{(i)}$. Если $\boldsymbol{x} \in \mathbb{R}^D$, $\boldsymbol{h} \in \mathbb{R}^d$ и $d < D$, то оптимальная ошибка реконструкции (при указанном выше выборе $\boldsymbol{\mu}$, $\boldsymbol{b}$, $\boldsymbol{V}$ и $\boldsymbol{W}$) равна

$$\min \mathbb{E}[\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2] = \sum_{i=d+1}^{D} \lambda_i. \tag{13.23}$$

Поэтому если ковариационная матрица имеет ранг d, то собственные значения от λ_{d+1} до λ_D равны 0 и ошибка реконструкции равна 0.

Далее, можно показать, что это же решение может быть получено максимизацией дисперсий элементов $\boldsymbol{h}$ при условии ортогональности $\boldsymbol{W}$, а не минимизацией ошибки реконструкции.

Линейные факторные модели относятся к простейшим порождающим моделям и к простейшим моделям, обучающимся представлению данных. Как линейные классификаторы и модели линейной регрессии можно обобщить на глубокие сети прямого распространения, так и линейные факторные модели обобщаются на сети-автокодировщики и глубокие вероятностные модели, которые решают те же задачи, но образуют гораздо более мощное и гибкое семейство моделей.

Глава 14

Автокодировщики

Автокодировщиком называется нейронная сеть, обученная пытаться скопировать свой вход в выход. На внутреннем уровне в ней имеется скрытый слой $\boldsymbol{h}$, который описывает **код**, используемый для представления входа. Можно считать, что сеть состоит из двух частей: функция кодирования $\boldsymbol{h} = f(\boldsymbol{x})$ и декодер, порождающий реконструкцию $\boldsymbol{r} = g(\boldsymbol{h})$. Эта архитектура показана на рис. 14.1. Если автокодировщику удается просто обучиться всюду сохранять тождество $g(f(\boldsymbol{x})) = \boldsymbol{x}$, то это не особенно полезно. На самом деле автокодировщики проектируются так, чтобы они не могли обучиться идеальному копированию. Обычно налагаются ограничения, позволяющие копировать только приближенно и только тот вход, который похож на обучающие данные. Поскольку модель заставляют расставлять подлежащие копированию аспекты данных по приоритетам, то часто она обучается полезным свойствам данных.

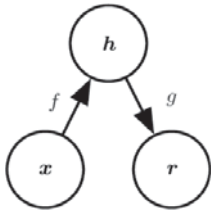

Рис. 14.1 ❖ Общая структура автокодировщика, отображающего вход $\boldsymbol{x}$ на выход $\boldsymbol{r}$ (называемый реконструкцией) через внутреннее представление, или код $\boldsymbol{h}$. Автокодировщик состоит из двух частей: кодировщик f (отображение $\boldsymbol{x}$ в $\boldsymbol{h}$) и декодер g (отображение $\boldsymbol{h}$ в $\boldsymbol{r}$)

В современных автокодировщиках идея кодировщика и декодера обобщена с детерминированных на стохастические отображения $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ и $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$.

Идея автокодировщиков исторически является частью ландшафта нейронных сетей уже несколько десятилетий (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994). Традиционно их использовали для понижения размерности и обучения признакам. Выявленные недавно теоретические связи между автокодировщиками и моделями с латентными переменными вывели автокодировщики на передний фронт порождающего моделирования, как мы увидим в главе 20. Автокодировщики можно рассматривать как частный случай сетей прямого распространения и обучать, применяя те же методы, обычно мини-пакетный градиентный спуск в направлении градиентов, вычисленных в ходе обратного распространения. Но, в отличие от общих сетей прямого распространения, автокодировщики можно обучать также с помощью

рециркуляции (Hinton and McClelland, 1988) – алгоритма обучения, основанного на сравнении активаций сети на оригинальных и реконструированных входных данных. Рециркуляция считается биологически более правдоподобным механизмом, чем обратное распространение, но редко применяется в приложениях машинного обучения.

14.1. Понижающие автокодировщики

Копирование входа в выход может показаться бесполезным делом, но, как правило, нас не интересует выход декодера. Вместо этого мы надеемся, что, обучая автокодировщик задаче копирования входа, мы получим код $\boldsymbol{h}$, обладающий полезными свойствами. Один из способов получить такие свойства – наложить ограничение, согласно которому размерность $\boldsymbol{h}$ должна быть меньше размерности $\boldsymbol{x}$. Такие автокодировщики называются **понижающими** (undercomplete). Обучение понижающего представления заставляет автокодировщик улавливать наиболее отличительные признаки обучающих данных.

Процесс обучения описывается просто как минимизация функции потерь

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))), \tag{14.1}$$

где функция L штрафует $g(f(\boldsymbol{x}))$ за непохожесть на $\boldsymbol{x}$; например, это может быть среднеквадратическая ошибка.

Если декодер – линейная функция, а L – среднеквадратическая ошибка, то понижающий автокодировщик обучается тому же подпространству, что PCA. В таком случае автокодировщик, обученный выполнять задачу копирования, в качестве побочного эффекта находит главное подпространство обучающих данных.

Следовательно, автокодировщики с нелинейными функциями кодирования f и декодирования g могут обучиться более мощным нелинейным обобщениям PCA. К сожалению, если разрешить слишком большую емкость кодировщика и декодера, то автокодировщик будет решать задачу копирования, не извлекая полезной информации о распределении данных. Теоретически можно вообразить автокодировщик с одномерным кодом, но очень мощным нелинейным кодировщиком, который обучится представлять каждый обучающий пример $\boldsymbol{x}^{(i)}$ кодом i. Декодер можно было бы обучить отображению этих целочисленных индексов обратно на значения конкретных обучающих примеров. Такой сценарий, на практике не встречающийся, ясно показывает, что автокодировщик, обученный копированию, может не собрать никакой полезной информации о наборе данных, если его емкость слишком велика.

14.2. Регуляризированные автокодировщики

Понижающие автокодировщики, для которых размерность кода меньше размерности входа, могут обучиться наиболее отличительным признакам распределения данных. Мы видели, что такие автокодировщики не обучаются ничему полезному, если емкость кодировщика и декодера слишком велика.

Похожая проблема возникает, если скрытый код имеет такую размерность, как вход, а также в случае **повышающего** (overcomplete) автокодировщика, когда размерность скрытого кода больше размерности входа. В этих случаях даже линейные кодировщик и декодер могут научиться копировать вход в выход, не узнав ничего полезного о распределении данных.

В идеале можно было бы успешно обучить любую архитектуру автокодировщика, выбрав размерность кода и емкость кодировщика и декодера, исходя из сложности моделируемого распределения. Такую возможность предоставляют регуляризированные автокодировщики. Вместо того чтобы ограничивать емкость модели, стремясь сделать кодировщик и декодер мелкими, а размер кода малым, регуляризированный автокодировщик использует такую функцию потерь, которая побуждает модель к приобретению дополнительных свойств, помимо способности копировать вход в выход. К числу таких свойств относятся разреженность представления, малая величина производной представления и устойчивость к шуму или отсутствию части входных данных. Регуляризированный автокодировщик может быть нелинейным и повышающим и тем не менее узнать что-то полезное о распределении данных, даже если емкость модели достаточно велика, чтобы обучить тривиальную тождественную функцию.

Помимо описанных ниже методов, которые наиболее естественно интерпретируются как регуляризированные автокодировщики, почти любая порождающая модель с латентными переменными и процедурой вывода (для вычисления латентных представлений по входу) может рассматриваться как некий вид автокодировщика. Существуют два подхода к порождающему моделированию, подчеркивающие такую связь с автокодировщиками: ведущие происхождение от машины Гельмгольца (Hinton et al., 1995b), например вариационный автокодировщик (раздел 20.10.3), и порождающие стохастические сети (раздел 20.12). Эти модели естественно обучаются повышающему кодированию входа с высокой емкостью и оказываются полезными даже без регуляризации кодирования. Созданные ими коды полезны, потому что модель обучалась приближенно максимизировать вероятность обучающих данных, а не копировать вход в выход.

14.2.1. Разреженные автокодировщики

Разреженным называется автокодировщик, для которого критерий обучения включает штраф разреженности $\Omega(\boldsymbol{h})$ на кодовом слое $\boldsymbol{h}$ в дополнение к ошибке реконструкции:

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}), \tag{14.2}$$

где $g(\boldsymbol{h})$ – выход декодера и обычно $\boldsymbol{h} = f(\boldsymbol{x})$ – выход кодировщика.

Разреженные автокодировщики чаще всего применяются для обучения признакам в интересах другой задачи, например классификации. Автокодировщик, регуляризированный с целью добиться разреженности, должен откликаться на уникальные статистические особенности набора данных, на котором обучался, а не просто выступать в роли тождественной функции. Поэтому обучение копированию со штрафом разреженности может дать модель, которая попутно обучилась каким-то полезным признакам.

Штраф $\Omega(\boldsymbol{h})$ можно рассматривать как регуляризирующий член, добавленный к сети прямого распространения, основная задача которой – копировать вход в выход (целевая функция для обучения без учителя) и, возможно, решать еще какую-то задачу обучения с учителем, зависящую от найденных разреженных признаков. В отличие от других регуляризаторов, например снижения весов, у этого регуляризатора нет очевидной байесовской интерпретации. Как было сказано в разделе 5.6.1, обучение со снижением весов и другими регуляризирующими штрафами можно ин-

терпретировать как аппроксимацию байесовского вывода максимумом апостериорной вероятности с добавлением регуляризирующего штрафа, который соответствует априорному распределению вероятности параметров модели. С этой точки зрения, регуляризированное максимальное правдоподобие соответствует максимизации $p(\boldsymbol{\theta} \mid \boldsymbol{x})$, что эквивалентно максимизации $\log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$. Член $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$ – обычное логарифмическое правдоподобие данных, а член $\log p(\boldsymbol{\theta})$, логарифмическое априорное распределение параметров, знаменует предпочтение определенным значениям $\boldsymbol{\theta}$. Этот взгляд на вещи описан в разделе 5.6. Для регуляризированных автокодировщиков такая интерпретация не годится, потому что регуляризатор зависит от данных и поэтому по определению не может считаться априорным распределением в формальном смысле слова. Однако мы по-прежнему можем считать, что регуляризирующие члены неявно выражают предпочтение некоторым функциям.

Вместо того чтобы считать штраф разреженности регуляризатором для копирования данных, мы можем рассматривать всю инфраструктуру разреженного автокодирования как обучение порождающей модели с латентными переменными, аппроксимирующее максимальное правдоподобие. Предположим, что имеется модель с видимыми переменными $\boldsymbol{x}$, латентными переменными $\boldsymbol{h}$ и явным совместным распределением $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h}) = p_{\text{model}}(\boldsymbol{h}) p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$. Мы называем $p_{\text{model}}(\boldsymbol{h})$ априорным распределением латентных переменных и считаем, что оно представляет априорную веру модели в то, что она увидит $\boldsymbol{x}$. Эта трактовка отличается от предыдущего употребления слова «априорный», которое обозначало распределение $p(\boldsymbol{\theta})$, описывающее наши гипотезы о параметрах модели еще до знакомства с обучающими данными. Логарифмическое правдоподобие можно представить в виде

$$\log p_{\text{model}}(\boldsymbol{x}) = \log \sum_{\boldsymbol{h}} p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x}). \tag{14.3}$$

Мы можем рассматривать автокодировщик как аппроксимацию этой суммы точечной оценкой для всего одного значения $\boldsymbol{h}$, имеющего высокую вероятность. Это похоже на порождающую модель разреженного кодирования (раздел 13.4), только $\boldsymbol{h}$ теперь является выходом параметрического кодировщика, а не результатом оптимизации, которая выводит наиболее вероятное значение $\boldsymbol{h}$. С этой точки зрения, при таком выборе $\boldsymbol{h}$ мы максимизируем функцию

$$\log p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x}) = \log p_{\text{model}}(\boldsymbol{h}) + \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h}). \tag{14.4}$$

Член $\log p_{\text{model}}(\boldsymbol{h})$ может индуцировать разреженность. Например, априорное распределение Лапласа

$$p_{\text{model}}(h_i) = \frac{\lambda}{2} e^{-\lambda |h_i|}, \tag{14.5}$$

соответствует штрафу разреженности по норме L^1 (сумма абсолютных величин). Действительно,

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i|, \tag{14.6}$$

$$-\log p_{\text{model}}(\boldsymbol{h}) = \sum_i \left(\lambda |h_i| - \log \frac{\lambda}{2} \right) = \Omega(\boldsymbol{h}) + \text{const}, \tag{14.7}$$

где постоянный член зависит только от λ и не зависит от $\boldsymbol{h}$. Обычно λ считается гиперпараметром, а постоянный член отбрасывается, потому что не влияет на обучение

параметров. Разреженность могут индуцировать и другие априорные распределения, например t-распределение Стьюдента. При таком взгляде на разреженность как результат влияния $p_{\text{model}}(\boldsymbol{h})$ на обучение с целью аппроксимации максимального правдоподобия штраф разреженности вообще не является регуляризирующим членом. Это просто следствие распределения латентных переменных модели. Такой взгляд дает еще один мотив для обучения автокодировщика: это способ приближенного обучения порождающей модели. И попутно выясняется еще одна причина полезности признаков, обученных автокодировщиком: они описывают латентные переменные, объясняющие входные данные.

В ранних работах по разреженным автокодировщикам (Ranzato et al., 2007a, 2008) исследовались различные формы разреженности и была предложена связь между штрафом разреженности и членом $\log Z$, который возникает в случае применения максимального правдоподобия к неориентированной вероятностной модели $p(\boldsymbol{x}) = (1/Z)\tilde{p}(\boldsymbol{x})$. Идея в том, что минимизация $\log Z$ не дает вероятностной модели достичь высокой вероятности всюду, а наложение ограничения разреженности на автокодировщик предотвращает низкую ошибку реконструкции всюду. В этом случае связь прослеживается на уровне интуитивного понимания общего механизма, а не математически строгого соответствия. Интерпретация штрафа разреженности как соответствия величине $\log p_{\text{model}}(\boldsymbol{h})$ в ориентированной модели $p_{\text{model}}(\boldsymbol{h})p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$ математически более очевидна.

Один из способов достижения настоящих нулей в коде $\boldsymbol{h}$ для разреженных (и шумоподавляющих) автокодировщиков предложен в работе Glorot et al. (2011b). Идея в том, чтобы использовать блоки линейной ректификации для порождения кодового слоя. Взяв априорное распределение, которое толкает представления в сторону нуля (например, штраф по норме L^1), можно косвенно управлять средним числом нулей в представлении.

14.2.2. Шумоподавляющие автокодировщики

Получить автокодировщик, который обучается чему-то полезному, можно не только путем добавления штрафа в функцию стоимости, но и изменив в ней член реконструкции ошибки.

Традиционно автокодировщики минимизируют некоторую функцию

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))), \tag{14.8}$$

где L – функция потерь, штрафующая $g(f(\boldsymbol{x}))$ за непохожесть на $\boldsymbol{x}$, например норма L^2 разности. Это побуждает $g \circ f$ быть просто тождественной функцией, если емкости g и f для этого достаточно.

Шумоподавляющий автокодировщик (denoising autoencoder – DAE) вместо этого минимизирует функцию

$$L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}}))), \tag{14.9}$$

где $\tilde{\boldsymbol{x}}$ – копия $\boldsymbol{x}$, искаженная каким-то шумом. Поэтому шумоподавляющий автокодировщик должен скорректировать искажение, а не просто скопировать свой вход.

Как показано в работах Alain and Bengio (2013) и Bengio et al. (2013c), обучение с целью шумоподавления принуждает f и g неявно обучиться структуре $p_{\text{data}}(\boldsymbol{x})$. Следовательно, шумоподавляющие автокодировщики – еще один пример того, как полезные свойства могут оказаться побочным результатом минимизации ошибки

реконструкции. Вместе с тем это пример того, как повышающую модель с высокой емкостью можно использовать в качестве автокодировщика, если позаботиться о том, чтобы она не обучилась тождественной функции. Шумоподавляющие автокодировщики более детально описаны в разделе 14.5.

14.2.3. Регуляризация посредством штрафования производных

Еще одна стратегия регуляризации автокодировщика подразумевает использование штрафа, как в разреженных автокодировщиках:

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}, \boldsymbol{x}), \tag{14.10}$$

но с Ω другого вида:

$$\Omega(\boldsymbol{h}, \boldsymbol{x}) = \lambda \sum_i \left\| \nabla_{\boldsymbol{x}} h_i \right\|^2. \tag{14.11}$$

Это понуждает модель обучить функцию, которая не сильно изменяется при малом изменении $\boldsymbol{x}$. Поскольку этот штраф применяется только к обучающим примерам, он заставляет автокодировщик обучиться признакам, улавливающим информацию об обучающем распределении.

Регуляризированный таким способом автокодировщик называется **сжимающим** (contractive autoencoder – CAE). Существуют теоретические связи между этим подходом и шумоподавляющими автокодировщиками, обучением многообразий и вероятностным моделированием. Сжимающие автокодировщики подробнее описаны в разделе 14.7.

14.3. Репрезентативная способность, размер слоя и глубина

Часто при обучении автокодировщиков используют кодировщик и декодер всего с одним слоем. Но это необязательно. На самом деле глубокие кодировщики и декодеры дают целый ряд преимуществ.

Напомним (см. раздел 6.4.1), что у глубоких сетей прямого распространения есть много достоинств. Поскольку автокодировщики – это сети прямого распространения, то все эти достоинства свойственны и им тоже. Более того, кодировщик и декодер по отдельности тоже являются сетями прямого распространения, поэтому каждая компонента автокодировщика может получить выигрыш от глубины.

Важное преимущество нетривиальной глубины состоит в том, что согласно универсальной теореме аппроксимации нейронной сетью прямого распространения хотя бы с одним скрытым слоем гарантированно можно аппроксимировать любую функцию (из весьма широкого класса) с произвольной точностью, при условии что количество скрытых блоков достаточно велико. Это означает, что автокодировщик с одним скрытым слоем способен представить тождественную функцию в области определения данных с произвольной точностью. Однако отображение входа на код мелкое, т. е. мы не можем наложить произвольных ограничений, например потребовать, чтобы код был разреженным. Глубокий автокодировщик, имеющий, по крайней мере, один дополнительный скрытый слой внутри самого кодировщика, может аппроксимировать любое отображение входа на код с произвольной точностью при наличии достаточного числа скрытых блоков.

Увеличение глубины может экспоненциально уменьшить вычислительную стоимость представления некоторых функций, а также объем данных, необходимых для обучения некоторых функций. Обзор преимуществ глубины в сетях прямого распространения см. в разделе 6.4.1.

Экспериментально показано, что глубокие автокодировщики достигают гораздо более высокой степени сжатия, чем соответствующие мелкие или линейные автокодировщики (Hinton and Salakhutdinov, 2006).

Общая стратегия обучения глубокого автокодировщика состоит в жадном предобучении глубокой архитектуры посредством обучения ряда мелких автокодировщиков, именно поэтому мы часто встречаем мелкие автокодировщики даже тогда, когда целью является обучение глубокого.

14.4. Стохастические кодировщики и декодеры

Автокодировщики – это просто сети прямого распространения. Те же функции потерь и типы выходных блоков, что применяются в традиционных сетях прямого распространения, можно использовать и в автокодировщиках.

Как было сказано в разделе 6.2.2.4, общая стратегия проектирования выходных блоков и функции потерь в сети прямого распространения заключается в том, чтобы определить выходное распределение $p(\boldsymbol{y} \mid \boldsymbol{x})$ и минимизировать отрицательное логарифмическое правдоподобие $-\log p(\boldsymbol{y} \mid \boldsymbol{x})$. В такой постановке $\boldsymbol{y}$ – это вектор целей, например меток классов.

В автокодировщике $\boldsymbol{x}$ является не только входом, но и целью. Но тем не менее мы можем применить тот же механизм, что и раньше. Можно считать, что декодер дает условное распределение $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$ при условии скрытого кода $\boldsymbol{h}$. Тогда можно обучить автокодировщик путем минимизации $-\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. Точная форма этой функции потерь будет зависеть от формы декодера. Как и в традиционных сетях прямого распространения, когда $\boldsymbol{x}$ принимает вещественные значения, мы обычно используем линейные выходные блоки для параметризации среднего нормального распределения. В этом случае отрицательное логарифмическое правдоподобие дает критерий среднеквадратической ошибки. Аналогично бинарным значениям $\boldsymbol{x}$ соответствует распределение Бернулли, параметры которого задаются сигмоидным выходным блоком, дискретным значениям $\boldsymbol{x}$ – softmax-распределение и т. д. Как правило, выходные переменные считаются условно независимыми при условии $\boldsymbol{h}$, чтобы это распределение вероятности было проще вычислить, но существуют методы, например выход в виде смеси распределений, которые позволяют моделировать коррелированный выход с приемлемыми вычислительными затратами.

Чтобы еще дальше отойти от сетей прямого распространения, мы можем обобщить понятие **функции кодирования** $f(\boldsymbol{x})$ на **распределение кодирования** $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$, как показано на рис. 14.2.

Любая модель с латентными переменными $p_{\text{model}}(\boldsymbol{h}, \boldsymbol{x})$ определяет стохастический кодировщик

$$p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x}) = p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x}) \tag{14.12}$$

и стохастический декодер

$$p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h}) = p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h}). \tag{14.13}$$

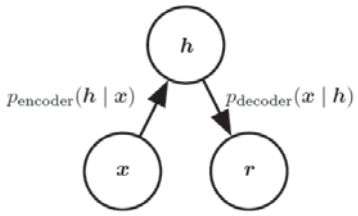

Рис. 14.2 ❖ Структура стохастического автокодировщика, в которой кодировщик и декодер – функции, содержащие шум, т. е. их выход можно рассматривать как выборку из распределения $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ для кодировщика и $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$ для декодера

В общем случае распределения кодировщика и декодера не обязательно являются условными распределениями, совместимыми с совместным распределением $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h})$. В работе Alain et al. (2015) показано, что обучение кодировщика и декодера как шумоподавляющего автокодировщика делает их асимптотически совместимыми (при достаточной емкости и количестве обучающих примеров).

14.5. Шумоподавляющие автокодировщики

Шумоподавляющим автокодировщиком (DAE) называется автокодировщик, который получает на входе искаженные данные и обучается предсказывать истинные, неискаженные данные.

Процедура обучения DAE показана на рис. 14.3. Мы вводим искажающий процесс $C(\tilde{\mathbf{x}} \mid \mathbf{x})$, который представляет условное распределение искаженных примеров $\tilde{\mathbf{x}}$ при условии истинных примеров $\mathbf{x}$. Затем автокодировщик обучается **распределению реконструкции** $p_{\text{reconstruct}}(\mathbf{x} \mid \tilde{\mathbf{x}})$, которое оценивается по обучающим парам $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ следующим образом:

1) выбрать обучающий пример $\boldsymbol{x}$ из обучающих данных;
2) выбрать искаженную версию $\tilde{\boldsymbol{x}}$ из $C(\tilde{\mathbf{x}} \mid \mathbf{x} = \boldsymbol{x})$;
3) использовать $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ в качестве обучающего примера для оценки распределения реконструкции автокодировщика $p_{\text{reconstruct}}(\boldsymbol{x} \mid \tilde{\boldsymbol{x}}) = p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$, где $\boldsymbol{h}$ – выход кодировщика $f(\tilde{\boldsymbol{x}})$, а p_{decoder} обычно определяется декодером $g(\boldsymbol{h})$.

Как правило, мы можем просто выполнить приближенную градиентную минимизацию (например, мини-пакетный градиентный спуск) отрицательного логарифмического правдоподобия $-\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$. Если кодировщик детерминирован, то шумоподавляющий автокодировщик является сетью прямого распространения, и для его обучения можно применить все те же методы, что для любой сети прямого распространения.

Таким образом, мы можем считать, что DAE применяет метод стохастического градиентного спуска к следующему математическому ожиданию:

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}(\mathbf{x})} \mathbb{E}_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}} \mid \mathbf{x})} \log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}})), \tag{14.14}$$

где $\hat{p}_{\text{data}}(\mathbf{x})$ – распределение обучающих данных.

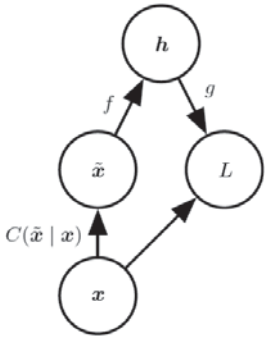

Рис. 14.3 ❖ Граф вычислений функции стоимости для шумоподавляющего автокодировщика, обученного реконструировать чистые данные $\boldsymbol{x}$ по искаженным $\tilde{\boldsymbol{x}}$. Это достигается путем минимизации потери $L = -\log p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h} = f(\tilde{\boldsymbol{x}}))$, где $\tilde{\boldsymbol{x}}$ – искаженная версия примера $\boldsymbol{x}$, полученная посредством заданного искажающего процесса $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$. Обычно p_{decoder} является факторным распределением, средние параметры которого порождаются сетью прямого распространения g

14.5.1. Сопоставление рейтингов

Сопоставление рейтингов (score matching) (Hyvärinen, 2005) – альтернатива максимальному правдоподобию. Этот метод дает состоятельную оценку распределений вероятности, поощряя модель иметь такой же **рейтинг** (score), как распределение данных на каждом обучающем примере $\boldsymbol{x}$. В этом контексте рейтингом является конкретное поле градиента:

$$\nabla_x \log p(\boldsymbol{x}). \tag{14.15}$$

Мы вернемся к обсуждению сопоставления рейтингов в разделе 18.4. Пока что достаточно понимать, что обучения поля градиента $\log p_{\text{data}}$ – один из способов обучиться структуре самого распределения p_{data}.

У шумоподавляющих автокодировщиков есть очень важное свойство: критерий их обучения (с условно нормальным $p(\boldsymbol{x} \mid \boldsymbol{h})$) таков, что автокодировщик обучается векторному полю $(g(f(\boldsymbol{x})) - \boldsymbol{x})$, являющемуся оценкой для рейтинга распределения данных. Это показано на рис. 14.4.

Шумоподавляющее обучение автокодировщиков конкретного вида (с сигмоидными скрытыми блоками и линейными блоками реконструкции) с использованием гауссова шума и среднеквадратической ошибки в качестве стоимости реконструкции эквивалентно (Vincent, 2011) обучению специального вида неориентированной вероятностной модели – ограниченной машины Больцмана (ОМБ) с гауссовыми видимыми блоками. Эта модель подробно описана в разделе 20.5.1; сейчас нам достаточно знать, что она дает явное распределение $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$. Если ОМБ обучается с применением **шумоподавляющего сопоставления рейтингов** (Kingma and LeCun, 2010), то алгоритм обучения эквивалентен шумоподавляющему обучению в соответствующем автокодировщике. При фиксированном уровне шума регуляризированное сопоставление рейтингов не является состоятельной оценкой, оно восстанавливает размытую версию распределения. Но если уровень шума стремится к 0, когда число примеров

стремится к бесконечности, то состоятельность восстанавливается. Шумоподавляющее сопоставление рейтингов обсуждается в разделе 18.5.

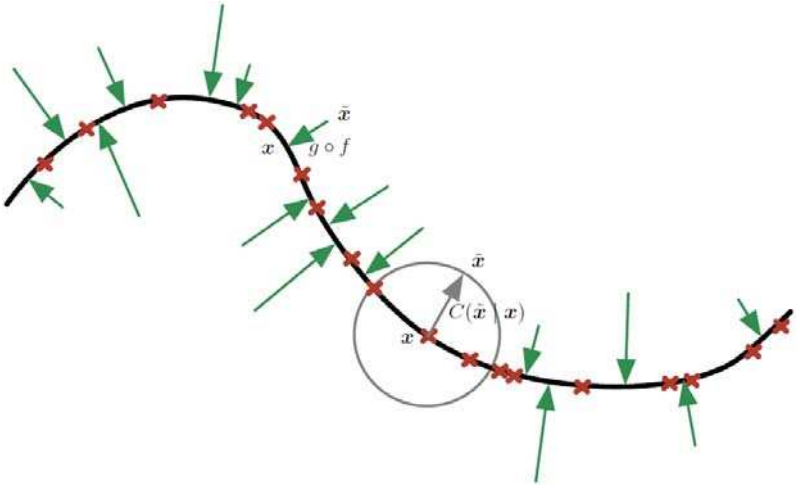

Рис. 14.4 ❖ Шумоподавляющий автокодировщик обучается отображать искаженные данные $\tilde{\boldsymbol{x}}$ на исходные $\boldsymbol{x}$. Обучающие примеры $\boldsymbol{x}$ обозначены красными крестиками, лежащими в окрестности многообразия низкой размерности, показанного жирной черной линией. Искажающий процесс $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ представлен серой окружностью с равновероятными искажениями. Серая стрелка показывает, как один обучающий пример преобразуется в результат одной выборки из искажающего процесса. Когда шумоподавляющий автокодировщик обучается минимизировать среднее квадратичных ошибок $\|g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}\|^2$, реконструкция $g(f(\tilde{\boldsymbol{x}}))$ оценивает $\mathbb{E}_{\mathbf{x},\tilde{\mathbf{x}} \sim p_{\text{data}}(\mathbf{x})C(\tilde{\mathbf{x}}|\mathbf{x})}[\mathbf{x} \mid \tilde{\boldsymbol{x}}]$. Вектор $g(f(\tilde{\boldsymbol{x}})) - \tilde{\boldsymbol{x}}$ направлен приблизительно в сторону ближайшей точки на многообразии, поскольку $g(f(\tilde{\boldsymbol{x}}))$ оценивает центр тяжести неискаженных точек $\boldsymbol{x}$, которые могли бы привести к $\tilde{\boldsymbol{x}}$. Таким образом, автокодировщик обучается векторному полю $g(f(\boldsymbol{x})) - \boldsymbol{x}$, обозначенному зелеными стрелками. Это векторное поле является оценкой рейтинга $\nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x})$ с точностью до множителя, равного среднеквадратической ошибке реконструкции

Существуют и другие связи между автокодировщиками и ОМБ. Сопоставление рейтингов в применении к ОМБ дает функцию стоимости, идентичную ошибке реконструкции в сочетании с регуляризирующим членом, аналогичным сжимающему штрафу CAE (Swersky et al., 2011). В работе Bengio and Delalleau (2009) показано, что градиент автокодировщика дает аппроксимацию обучения ОМБ методом сопоставительного расхождения (contrastive divergence).

Для непрерывных $\boldsymbol{x}$ шумоподавляющий критерий с гауссовым искажением и распределением реконструкции дает оценку рейтинга, применимую к общим параметризациям кодировщика и декодера (Alain and Bengio, 2013). Это означает, что общей архитектурой кодировщик-декодер можно воспользоваться для оценивания рейтинга, если проводить обучение с критерием квадратичной ошибки

$$\|g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}\|^2 \tag{14.16}$$

и искажающим процессом

$$C(\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}} \mid \boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}}; \mu = \boldsymbol{x}, \Sigma = \sigma^2 I) \tag{14.17}$$

с дисперсией шума σ^2. Принцип работы иллюстрируется на рис. 14.5.

В общем случае не гарантируется, что реконструкция $g(f(\boldsymbol{x}))$ минус вход $\boldsymbol{x}$ соответствует градиенту хоть какой-нибудь функции, не говоря уже о рейтинге. Именно поэтому ранние результаты (Vincent, 2011) специализированы для конкретных параметризаций, когда $g(f(\boldsymbol{x})) - \boldsymbol{x}$ можно получить в виде производной какой-то другой функции. В работе Kamyshanska and Memisevic (2015) результат Vincent (2011) обобщен путем идентификации такого семейства мелких автокодировщиков, что $g(f(\boldsymbol{x})) - \boldsymbol{x}$ соответствует рейтингу для всех членов этого семейства.

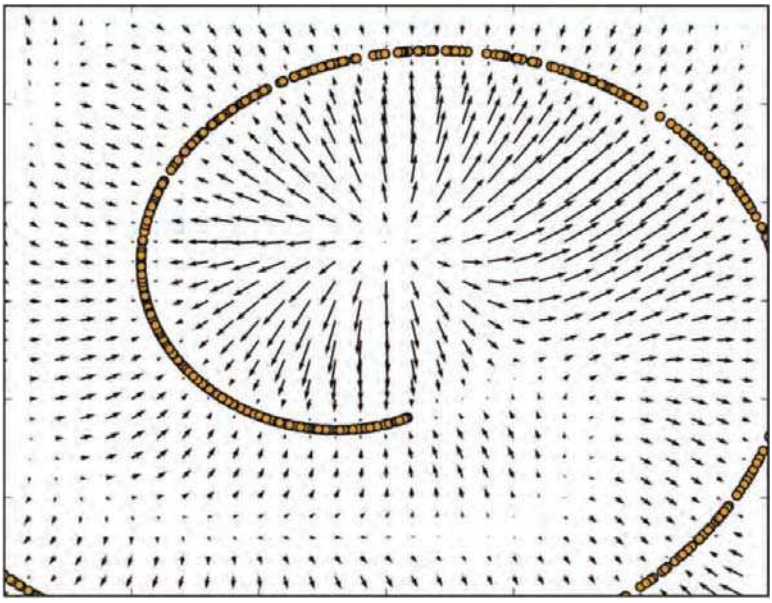

Рис. 14.5 ❖ Обученное шумоподавляющим автокодировщиком векторное поле вокруг одномерного искривленного многообразия, в окрестности которого сконцентрированы двумерные данные. Каждая стрелка по длине пропорциональна реконструкции минус входной вектор автокодировщика и направлена в сторону увеличения вероятности в соответствии с неявной оценкой распределения вероятности. Векторное поле обращается в нуль в точках максимума и минимума оцененной функции плотности (на многообразии). Так, отрезок спирали образует одномерное многообразие соединенных друг с другом локальных максимумов. Локальные минимумы находятся вблизи середины разрыва между двумя отрезками. Если норма ошибки реконструкции (пропорциональная длинам стрелок) велика, то вероятность можно значительно повысить, двигаясь в направлении стрелки; такое бывает в основном на участках низкой вероятности. Автокодировщик отображает такие точки с низкой вероятностью на реконструкции, имеющие более высокую вероятность. Там, где вероятность максимальна, стрелка укорачивается, поскольку реконструкция становится более точной. Рисунок взят из работы Alain and Bengio (2013) с разрешения авторов

До сих пор мы описывали только, как шумоподавляющий автокодировщик обучается представлять распределение вероятности. В более общем случае можно использовать автокодировщик как порождающую модель и делать выборку из этого распределения. Это описывается в разделе 20.11.

14.5.1.1. Историческая справка

Идея использовать МСП для шумоподавления восходит к работам LeCun (1987) и Gallinari et al. (1987). В работе Behnke (2001) рекуррентные сети использовались для очистки изображений от шумов. В некотором смысле шумоподавляющие автокодировщики – это просто МСП, обученные шумоподавлению.

Однако название «шумоподавляющий автокодировщик» относится к модели, которая должна не просто обучиться очищать вход от шумов, но и найти хорошее внутреннее представление в качестве побочного эффекта. Эта идея возникла много позже (Vincent et al., 2008, 2010). Затем обученное представление можно использовать для предобучения более глубокой сети без учителя или с учителем. Как и в случае разреженного кодирования, разреженных, сжимающих и прочих регуляризированных автокодировщиков, мотивом для разработки шумоподавляющих автокодировщиков было стремление обучить кодировщики очень высокой емкости, не дав в то же время кодировщику и декодеру обучиться бесполезной тождественной функции.

Еще до появления современных DAE в работе Inayoshi and Kurita (2005) исследовалось достижение части тех же целей некоторыми из описанных выше методов. Подход авторов был основан на минимизации ошибки реконструкции в дополнение к целевой функции при внесении шума в скрытый слой обучаемого с учителем МСП, а целью было улучшить обобщаемость за счет введения ошибки реконструкции и привнесенного шума. Но они применяли линейный кодировщик и не могли обучать столь же мощные семейства функций, как современные DAE.

14.6. Обучение многообразий с помощью автокодировщиков

Как и во многих других алгоритмах машинного обучения, в автокодировщиках используется гипотеза о концентрации данных в окрестности многообразия низкой размерности или небольшого множества таких многообразий (см. раздел 5.11.3). В некоторых алгоритмах эта идея ограничена лишь обучением функций, которые корректно ведут себя на многообразии, но поведение может оказаться необычным, если предъявить пример, лежащий вне многообразия. Автокодировщики идут дальше и стремятся обучиться структуре многообразия.

Чтобы понять, как они это делают, необходимо познакомиться с некоторыми характеристиками многообразий.

Важной характеристикой является множество **касательных плоскостей**. Касательная плоскость в точке $\boldsymbol{x}$ на d-мерном многообразии определяется d базисными векторами в локальных направлениях допустимых изменений. Как показано на рис. 14.6, эти локальные направления описывают, какие возможны бесконечно малые изменения $\boldsymbol{x}$, не выводящие за пределы многообразия.

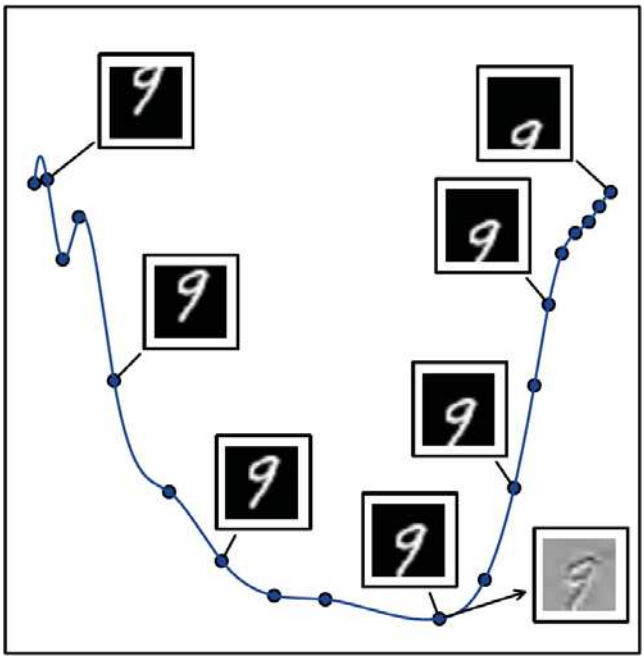

Рис. 14.6 ❖ Иллюстрация понятия касательной гиперплоскости. Здесь создается одномерное многообразие в 784-мерном пространстве. Мы берем изображение из набора данных MNIST, содержащее 784 пикселя, и преобразуем его, выполняя параллельный перенос по вертикали. Величина переноса определяет координату вдоль одномерного многообразия, которая описывает искривленный путь в пространстве изображения. На графике показано несколько точек на этом многообразии. Для наглядности мы спроецировали многообразие на двумерное пространство методом PCA. У n-мерного многообразия в каждой точке имеется n-мерная касательная плоскость. Эта плоскость имеет с многообразием ровно одну общую точку и ориентирована параллельно его поверхности в этой точке. Она определяет направления, в которых можно двигаться, оставаясь на многообразии. У данного одномерного многообразия касательная плоскость вырождается в прямую. Мы привели пример касательной прямой в одной точке вместе с изображением, показывающим, как ее направление выглядит в пространстве изображения. Серым цветом обозначены пиксели, которые не изменяются при движении вдоль касательной, белым – пиксели, которые становятся ярче, а черным – те, что становятся темнее

Любая процедура обучения автокодировщика включает компромисс между двумя стремлениями.

1. Обучить такое представление $\boldsymbol{h}$ обучающего примера $\boldsymbol{x}$, чтобы $\boldsymbol{x}$ можно было приближенно восстановить по $\boldsymbol{h}$ с помощью декодера. Тот факт, что $\boldsymbol{x}$ выбирается из обучающих данных, критически важен, поскольку означает, что автокодировщик не обязан успешно реконструировать входы, не вероятные с точки зрения порождающего данные распределения.
2. Удовлетворить ограничение или регуляризирующий штраф. Это может быть архитектурное ограничение, ограничивающее емкость автокодировщика, или ре-

гуляризирующий член, прибавленный к стоимости реконструкции. Смысл в любом случае – отдавать предпочтение решениям, менее чувствительным к входу.

Очевидно, что поодиночке оба стремления бесполезны: ни копирование входа в выход, ни полное игнорирование входа нам ни к чему. Но вместе они становятся осмысленными, потому что вынуждают скрытое представление уловить информацию о структуре порождающего данные распределения. Важный принцип заключается в том, что автокодировщик может ограничиться представлением *только тех изменений, которые необходимы для реконструкции обучающих примеров*. Если порождающее распределение концентрируется вблизи многообразия низкой размерности, то получатся представления, которые неявно улавливают локальную систему координат этого многообразия: лишь изменения, касательные к многообразию в точке $\boldsymbol{x}$, должны соответствовать изменениям $\boldsymbol{h} = f(\boldsymbol{x})$. Следовательно, кодировщик обучается такому отображению из пространства входов $\boldsymbol{x}$ в пространство представления, которое чувствительно только к изменениям вдоль направлений, касательных к многообразию, и нечувствительно к изменениям вдоль ортогональных к многообразию направлений.

На рис. 14.7 приведен одномерный пример, показывающий, что, сделав функцию реконструкции нечувствительной к возмущениям входа в окрестности данных, мы заставили автокодировщик восстановить структуру многообразия.

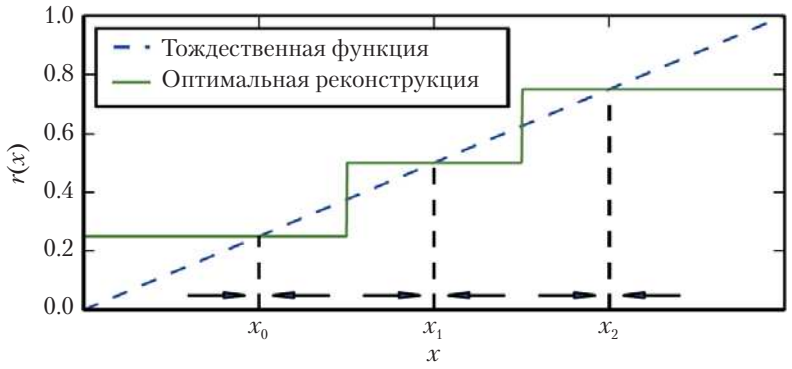

Рис. 14.7 ❖ Если автокодировщик обучает функцию реконструкции, инвариантную к малым возмущениям в окрестности входных точек, то он улавливает структуру многообразия, на котором лежат данные. В данном случае структурой является набор 0-мерных многообразий. Штриховая диагональная прямая обозначает тождественную целевую функцию, подлежащую реконструкции. Оптимальная функция реконструкции пересекает график тождественной функции в каждой точке данных. Горизонтальные стрелки в нижней части рисунка обозначают вектор направления реконструкции $r(\boldsymbol{x}) - \boldsymbol{x}$ в пространстве входов и всегда указывают в сторону ближайшего «многообразия» (единственной точки в одномерном случае). Шумоподавляющий автокодировщик пытается сделать производную функции реконструкции $r(\boldsymbol{x})$ малой в окрестности входных точек. Сжимающий автокодировщик делает то же самое для кодировщика. Хотя производную $r(\boldsymbol{x})$ понуждают быть малой вблизи входных точек, между ними она может быть большой. Пространство между входными точками соответствует области между многообразиями, где функция реконструкции должна иметь большую производную, чтобы вернуть искаженные точки на многообразие

Чтобы понять, почему автокодировщики полезны для обучения многообразий, поучительно сравнить их с другими подходами. Для характеристики многообразия обычно обучают **представление** точек данных на многообразии (или вблизи него). Для конкретного примера такое представление называют также его погружением. Как правило, оно описывается вектором более низкой размерности, чем у объемлющего пространства, подмножеством которого является многообразие. Некоторые алгоритмы (обсуждаемые ниже непараметрические алгоритмы обучения многообразий) непосредственно обучают погружение для каждого обучающего примера, тогда как другие обучают более общее отображение, иногда называемое кодировщиком, или функцией представления, которое переводит любую точку объемлющего пространства (пространства входов) в ее погружение.

Обучение многообразий по большей части включает в себя процедуры обучения без учителя, пытающиеся эти многообразия уловить. В большинстве ранних работ по обучению нелинейных многообразий акцент ставился на **непараметрические** методы, основанные на **графе ближайших соседей**. В этом графе имеется одна вершина для каждого обучающего примера и ребра, соединяющие ближайших соседей. Эти методы (Schölkopf et al., 1998; Roweis and Saul, 2000; Tenenbaum et al., 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004; Hinton and Roweis, 2003; van der Maaten and Hinton, 2008) связывают с каждой вершиной касательную плоскость, натянутую на направления изменения, определяемые векторами разностей между примером и его соседями, как показано на рис. 14.8. Глобальную систему координат можно затем получить с помощью оптимизации или путем решения системы линейных уравнений. На рис. 14.9 показано, как можно замостить многообразие большим числом локально линейных «блинов», напоминающих гауссианы, плоские в касательных направлениях.

Фундаментальная трудность таких локальных непараметрических подходов к обучению многообразий описана в работе Bengio and Monperrus (2005): если многообразие не слишком гладкое (имеет много пиков, впадин и скручиваний), то для покрытия всех вариаций может понадобиться очень много обучающих примеров, но шансов обобщения на ранее не предъявлявшиеся вариации все равно не будет. Действительно, эти методы способны обобщить форму многообразия только путем интерполяции соседних примеров. К сожалению, многообразия, встречающиеся в задачах ИИ, могут иметь очень сложную структуру, которую трудно уловить одной лишь локальной интерполяцией. Рассмотрим пример многообразия, полученного в результате параллельных переносов, на рис. 14.6. Если мы наблюдаем только одну координату входного вектора x_i по мере переноса изображения, то будем видеть максимум или минимум этой координаты всякий раз, как в яркости изображения встречается пик или впадина. Иными словами, сложность паттернов яркости в исходном изображении определяет сложность многообразий, порождаемых его простыми преобразованиями. Это наводит на мысль об использовании распределенных представлений и глубокого обучения для улавливания структуры многообразия.

14.7. Сжимающие автокодировщики

Сжимающий автокодировщик (CAE) (Rifai et al., 2011a,b) вводит явный регуляризирующий член для кода $\boldsymbol{h} = f(\boldsymbol{x})$, поощряющий за небольшую величину производных f:

$$\Omega(\boldsymbol{h}) = \lambda \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 . \tag{14.18}$$

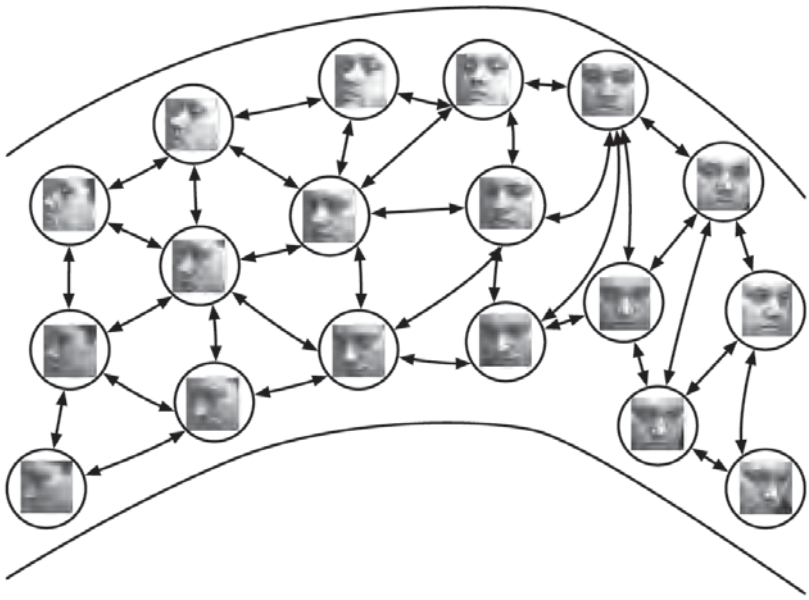

Рис. 14.8 ❖ Непараметрические процедуры обучения многообразий строят граф ближайших соседей, вершины которого представляют обучающие примеры, а ориентированные ребра – связи с ближайшими соседями. Существуют различные процедуры, позволяющие получить касательную плоскость, ассоциированную с соседством в этом графе, а также систему координат, которая ассоциирует каждый обучающий пример с положением вещественного вектора, или **погружением**. Такое представление обобщается на новые примеры с помощью интерполяции. Если число примеров достаточно велико для покрытия кривизны и скручивания многообразия, то такие методы работают хорошо. Изображения взяты из набора данных QMUL Multiview Face Dataset (Gong et al., 2000)

Штраф $\Omega(\boldsymbol{h})$ равен квадрату нормы Фробениуса (сумма квадратов элементов) матрицы Якоби, состоящей из частных производных функции кодирования.

Существует связь между шумоподавляющим и сжимающим автокодировщиками: в работе Alain and Bengio (2013) показано, что в пределе малого гауссова входного шума ошибка реконструкции шумоподавляющего автокодировщика эквивалентна сжимающему штрафу в функции реконструкции, которая отображает $\boldsymbol{x}$ в $\boldsymbol{r} = g(f(\boldsymbol{x}))$. Иными словами, шумоподавляющий автокодировщик понуждает функцию реконструкции сопротивляться малым, но конечным возмущениям входа, тогда как сжимающий автокодировщик заставляет функцию выделения признаков сопротивляться бесконечно малым возмущениям входа. Когда сжимающий штраф на основе матрицы Якоби применяется для предобучения признаков $f(\boldsymbol{x})$ с целью последующего использования в классификаторе, наилучшая верность классификации обычно получается, если применять штраф к $f(\boldsymbol{x})$, а не к $g(f(\boldsymbol{x}))$. Сжимающий штраф в $f(\boldsymbol{x})$ также имеет тесные связи с сопоставлением рейтингов, которое обсуждалось в разделе 14.5.1.

Термин **сжимающий** объясняется тем, как CAE деформирует пространство. Поскольку CAE обучен противиться возмущениям входа, то поощряется отображение окрестности входной точки в меньшую окрестность выходной точки. Это можно рассматривать как сжатие окрестности входной точки.

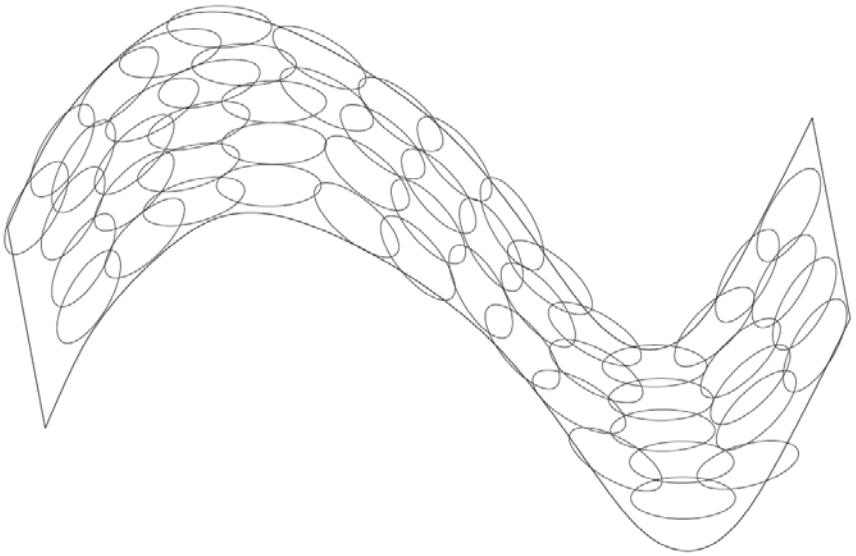

Рис. 14.9 ❖ Если касательные плоскости (см. рис. 14.6) в каждой точке известны, то из них можно образовать глобальную систему координат, или функцию плотности. Каждую локальную «плитку» можно рассматривать как локальную евклидову систему координат или как локально плоское нормальное распределение («блин») с очень малой дисперсией в направлениях, ортогональных блину, и очень большой в направлениях, определяющих систему координат блина. Смесь таких нормальных распределений дает оценку функции плотности, как в методе окна Парзена для многообразий (Vincent and Bengio, 2003), или его нелокальном варианте, основанном на нейронной сети (Bengio et al., 2006c)

Уточним, что CAE является сжимающим только локально – все возмущения точки $\boldsymbol{x}$ обучающего набора отображаются в малую окрестность $f(\boldsymbol{x})$. Глобально образы $f(\boldsymbol{x})$ и $f(\boldsymbol{x}')$ точек $\boldsymbol{x}$ и $\boldsymbol{x}'$ могут отстоять друг от друга дальше, чем исходные точки. Вполне может случиться, что между многообразиями или вдали от них функция f будет не сжимающей, а, наоборот, расширяющей (см., например, что происходит в тривиальном одномерном примере на рис. 14.7). Если штраф $\Omega(\boldsymbol{h})$ применяется к сигмоидным блокам, то для сжатия якобиана достаточно потребовать, чтобы эти блоки асимптотически приближались к 0 или 1. Тогда штраф поощряет CAE кодировать входные точки экстремальными значениями сигмоиды, что можно интерпретировать как двоичный код. Также гарантируется, что CAE будет рассредоточивать кодовые значения по большей части гиперкуба, покрываемого его сигмоидными скрытыми блоками.

Мы можем рассматривать матрицу Якоби $\boldsymbol{J}$ в точке $\boldsymbol{x}$ как аппроксимацию нелинейного кодировщика $f(\boldsymbol{x})$ линейным оператором. Это позволяет формализовать слово «сжимающий». В теории линейных операторов говорят, что линейный оператор сжимающий, если норма $\boldsymbol{Jx}$ меньше или равна 1 для всех векторов $\boldsymbol{x}$ с единичной нормой. Иначе говоря, $\boldsymbol{J}$ сжимающий, если он стягивает единичную сферу. Можно считать, что CAE штрафует норму Фробениуса локальной линейной аппроксимации $f(\boldsymbol{x})$ в каждой точке $\boldsymbol{x}$ обучающего набора, стремясь сделать все такие локальные линейные операторы сжимающими.

Как описано в разделе 14.6, регуляризированные автокодировщики обучают многообразия, балансируя под действием двух противоположно направленных сил. В случае CAE в роли таких сил выступают ошибка реконструкции и сжимающий штраф $\Omega(\boldsymbol{h})$. Одна лишь ошибка реконструкции поощряла бы CAE обучиться тождественной функции, а один лишь сжимающий штраф – обучиться признакам, постоянным относительно $\boldsymbol{x}$. В результате компромисса между тем и другим получается автокодировщик, у которого производные $\partial f(\boldsymbol{x})/\partial \boldsymbol{x}$ в основном очень малы. И лишь для немногих скрытых блоков, которые соответствуют небольшому числу направлений во входных данных, производные могут быть велики.

Цель CAE – обучиться структуре многообразия данных. В направлениях $\boldsymbol{x}$ с большими значениями $\boldsymbol{Jx}$ вектор $\boldsymbol{h}$ быстро изменяется, поэтому они, вероятно, являются направлениями, аппроксимирующими касательные плоскости к многообразию. Эксперименты, описанные в работе Rifai et al. (2011a,b), показывают, что обучение CAE приводит к тому, что в большинстве своем сингулярные числа $\boldsymbol{J}$ по абсолютной величине меньше 1, т. е. являются сжимающими. Но некоторые сингулярные числа все же оказываются больше 1, поскольку штраф за ошибку реконструкции поощряет CAE кодировать направления с наибольшей локальной дисперсией. Направления, соответствующие наибольшим сингулярным числам, интерпретируются как касательные направления, обученные сжимающим автокодировщиком. В идеале они должны соответствовать реальной вариативности данных. Например, в случае применения к изображениям CAE должен обучиться касательным векторам, которые показывают, как изменяется изображение, когда присутствующие в нем объекты постепенно меняют расположение, как на рис. 14.6. Визуализация экспериментально полученных сингулярных векторов, похоже, действительно соответствует осмысленным преобразованиям входного изображения, как видно по рис. 14.10.

С критерием регуляризации CAE возникает одна практическая проблема: его вычисление в случае автокодировщика с одним скрытым слоем обходится дешево, но становится гораздо дороже, когда слоев больше. В работе Rifai et al. (2011a) применена следующая стратегия – последовательность однослойных автокодировщиков обучается так, чтобы каждый следующий обучался реконструировать скрытый слой предыдущего. Их композиция и образует глубокий автокодировщик. Поскольку каждый слой локально сжимающий, то и глубокий автокодировщик тоже будет сжимающим. Результат получается не таким же, как при совместном обучении всей архитектуры со штрафом на якобиан глубокой модели, но многие желательные качественные характеристики улавливаются.

Еще одна практическая проблема состоит в том, что применение сжимающего штрафа может давать бесполезные результаты, если не задать какого-то масштаба декодера. Например, действие кодировщика может заключаться в умножении входа на небольшую константу ε, а действие декодера – в делении кода на ε. Когда ε стремится к 0, кодировщик устремляет сжимающий штраф $\Omega(\boldsymbol{h})$ к 0, так ничего и не узнав о распределении. А тем временем декодер демонстрирует идеальную реконструкцию. В работе Rifai et al. (2011a) для предотвращения такой ситуации веса f и g связываются. И f, и g – стандартные слои нейронной сети, состоящие из аффинного преобразования с последующей поэлементной нелинейностью, поэтому можно просто сделать матрицу весов g транспонированной к матрице весов f.

Рис. 14.10 ❖ Касательные векторы к многообразию, оцененные локальным методом главных компонент (PCA) и сжимающим автокодировщиком. Позиция на многообразии определяется входным изображением собаки, взятым из набора данных CIFAR-10. Для оценки касательных векторов используются первые сингулярные векторы матрицы Якоби $\partial \boldsymbol{h}/\partial \boldsymbol{x}$ отображения входа на код. Хотя и PCA, и CAE могут найти локальные касательные, CAE дает более точные оценки на ограниченном объеме обучающих данных, поскольку в нем присутствует разделение параметров между разными позициями, совместно использующими некоторое подмножество активных скрытых блоков. Касательные направления, найденные CAE, обычно соответствуют перемещающимся или изменяющимся частям объекта (скажем, голове или лапам). Изображение взято из работы Rifai et al. (2011c) с разрешения авторов

14.8. Предсказательная разреженная декомпозиция

Предсказательная разреженная декомпозиция (ПРД) (predictive sparse decomposition – PSD) – гибридная модель, в которой сочетаются разреженное кодирование и параметрические автокодировщики (Kavukcuoglu et al., 2008). Параметрический кодировщик обучается предсказывать результат итеративного вывода. ПРД применялась к обучению признаков без учителя для распознавания объектов в изображениях и видео (Kavukcuoglu et al., 2009, 2010; Jarrett et al., 2009; Farabet et al., 2011), а также к обработке звуковой информации (Henaff et al., 2011). Модель включает кодировщик $f(\boldsymbol{x})$ и декодер $g(\boldsymbol{h})$ – оба параметрические. В процессе обучения $\boldsymbol{h}$ контролируется алгоритмом оптимизации. Обучение заключается в минимизации

$$\|\boldsymbol{x} - g(\boldsymbol{h})\|^2 + \lambda|\boldsymbol{h}|_1 + \gamma\|\boldsymbol{h} - f(\boldsymbol{x})\|^2. \tag{14.19}$$

Как в случае разреженного кодирования, алгоритм обучения чередует минимизацию относительно $\boldsymbol{h}$ с минимизацией относительно параметров модели. Минимизация относительно $\boldsymbol{h}$ производится быстро, потому что $f(\boldsymbol{x})$ дает хорошее начальное значение $\boldsymbol{h}$, а функция стоимости налагает на $\boldsymbol{h}$ ограничение – оставаться близко к $f(\boldsymbol{x})$. Простым градиентным спуском можно получить разумные значения $\boldsymbol{h}$ всего за десять шагов.

Процедура обучения в ПРД не сводится к обучению сначала модели разреженного кодирования, а затем обучению $f(\boldsymbol{x})$ для предсказания значений признаков разреженного кодирования. На самом деле эта процедура регуляризирует декодер, так чтобы использовались параметры, для которых $f(\boldsymbol{x})$ может вывести хорошие значения кода.

Предсказательная разреженная декомпозиция – пример **обученного приближенного вывода**. В разделе 19.5 мы вернемся к этой теме. Из материала, изложенного в главе 19, следует, что ПРД можно интерпретировать как обучение ориентированной вероятностной модели разреженного кодирования путем максимизации нижней границы логарифмического правдоподобия модели.

В практических приложениях ПРД итеративная оптимизация используется только на этапе обучения. Параметрический кодировщик f применяется для вычисления обученных признаков после развертывания модели. Вычисление f обходится недорого, по сравнению с выводом $\boldsymbol{h}$ методом градиентного спуска. Поскольку f – дифференцируемая параметрическая функция, то модели ПРД можно объединить и использовать для инициализации глубокой сети, обучаемой по другому критерию.

14.9. Применения автокодировщиков

Автокодировщики успешно применялись в задачах понижения размерности и информационного поиска. Понижение размерности было одним из первых приложений обучения представлений и глубокого обучения, а также одним из ранних стимулов к изучению автокодировщиков. Например, в работе Hinton and Salakhutdinov (2006) был обучен стек ОМБ, а затем их веса использовались для инициализации глубокого автокодировщика с постепенно уменьшающимися скрытыми слоями, так что в последнем слое было всего 30 блоков. Получившийся код давал меньшую ошибку реконструкции, чем PCA, в 30 направлениях, а обученное представление оказалось проще качественно интерпретировать и связать с объясняющими категориями, которые проявлялись в виде четко разделенных кластеров.

Представления низкой размерности могут повысить качество во многих задачах, в т. ч. классификации. Чем меньше пространство, тем меньше памяти занимает модель и тем быстрее производятся расчеты. Многие способы понижения размерности помещают семантически похожие примеры рядом, как замечено в работах Salakhutdinov and Hinton (2007b) и Torralba et al. (2008). Информация, сохраняемая при отображении в пространство меньшей размерности, способствует лучшей обобщаемости.

Особенно сильно выигрывает от понижения размерности **информационный поиск** – задача об отыскании в базе данных записей, похожих на указанную в запросе. Дополнительный выигрыш связан с тем, что в некоторых видах пространств низкой размерности поиск может оказаться чрезвычайно эффективен. Точнее, если обучить алгоритм понижения размерности порождать двоичный код низкой размерности, то мы сможем поместить всю базу в хэш-таблицу, которая отображает двоичные кодовые векторы на полные записи. Тогда результатом информационного поиска будет множество всех записей базы данных с таким же двоичным кодом, как у запрошенной. Мы можем также очень эффективно искать чуть менее похожие данные, для чего нужно просто инвертировать отдельные биты в кодовом представлении запроса. Такой подход к информационному поиску посредством понижения размерности и бинаризации называется **семантическим хэшированием** (Salakhutdinov and Hinton, 2007b, 2009b), он успешно применялся к поиску как текстов (Salakhutdinov and Hinton, 2007b, 2009b), так и изображений (Torralba et al., 2008; Weiss et al., 2008; Krizhevsky and Hinton, 2011).

Для порождения двоичных кодов семантического хэширования обычно используется функция кодирования с сигмоидными блоками в последнем слое. Эти блоки

необходимо обучить так, чтобы они были асимптотически близки к 0 или к 1 для всех входных значений. Один из способов добиться этого – внести аддитивный шум перед сигмоидной нелинейностью во время обучения. Абсолютная величина шума должна возрастать со временем. Чтобы противостоять шуму и сохранить как можно больше информации, сеть должна увеличивать величину входов сигмоидной функции до наступления насыщения.

Идея обучения хэш-функции исследовалась в нескольких направлениях, в т. ч. для обучения оптимизирующих потерю представлений, которые в большей мере ориентированы на поиск близких примеров в хэш-таблице (Norouzi and Fleet, 2011).

Глава 15

Обучение представлений

В этой главе мы сначала обсудим, что значит обучить представление и чем понятие представления может быть полезно при проектировании глубоких архитектур. Мы рассмотрим, как алгоритмы обучения разделяют статистическую силу между разными задачами, в т. ч. поговорим об использовании обучения без учителя для решения задач обучения с учителем. Разделяемые представления полезны для обработки нескольких модальностей, или доменов, а также для передачи полученных в ходе обучения знаний задачам, для которых примеров мало или нет вовсе, зато существует представление. Наконец, мы вернемся назад и порассуждаем о том, почему обучение представлений оказалось столь успешным, начав с теоретических достоинств распределенных (Hinton et al., 1986) и глубоких представлений и закончив более общей идеей объясняющих предположений о процессе порождения данных и, в частности, об истинных причинах наблюдаемых данных.

Многие задачи обработки информации могут оказаться очень простыми или очень трудными в зависимости от представления информации. Этот общий принцип действует и в повседневной жизни, и в информатике вообще, и в машинном обучении в частности. Например, любой человек без труда разделит 210 на 6 с помощью деления в столбик. Но задача становится куда труднее, если числа записать в римской нотации. Если предложить современному человеку разделить CCX на VI, то он, скорее всего, сначала преобразует римские цифры в арабскую позиционную нотацию, допускающую деление в столбик. Мы можем количественно оценить асимптотическое время выполнения различных операций при использовании подходящего и неподходящего представлений. Так, вставка числа в нужную позицию отсортированного списка занимает время $O(n)$, если в качестве представления выбран связный список, и только $O(\log n)$, если выбрано красно-черное дерево.

А чем одно представление лучше другого в контексте машинного обучения? Вообще говоря, хорошим является представление, которое упрощает последующее обучение. Выбор представления обычно зависит от задачи обучения.

Сети прямого распространения, обучаемые с учителем, можно рассматривать как своего рода обучение представления. Точнее говоря, последний слой сети обычно является линейным классификатором, например softmax-регрессией. А вся остальная сеть формирует для этого классификатора представление. Обучение с учителем естественно создает в каждом скрытом слое представление с такими свойствами, которые делают классификацию проще (и чем ближе к верхнему слою, тем это вернее).

Например, классы, которые не были линейно разделимыми на входных признаках, могут стать таковыми в последнем скрытом слое. В принципе, последний слой может быть моделью любого вида, например классификатором по ближайшему соседу (Salakhutdinov and Hinton, 2007a). Признаки в предпоследнем слое должны обучиться различным свойствам в зависимости от типа последнего слоя.

Обучение сетей прямого распространения с учителем не налагает явных условий на обученные промежуточные признаки. Другие алгоритмы обучения представлений нередко проектируются так, что форма представления задается явно. Предположим, к примеру, что мы хотим обучить представление, упрощающее оценку плотности. Легче поддаются моделированию распределения с большей степенью независимости, поэтому мы могли бы спроектировать целевую функцию, поощряющую независимость элементов вектора представления $\boldsymbol{h}$. Как и у сетей с учителем, у алгоритмов глубокого обучения без учителя есть главная цель обучения, но в качестве побочного эффекта они обучаются некоторому представлению. Вне зависимости от способа получения это представление можно использовать для решения другой задачи. Или же можно вместе обучить несколько моделей (одни с учителем, другие без), разделяющих общее внутреннее представление. В большинстве проблем обучения представления приходится выбирать между сохранением как можно более полной информации о входе и приобретением полезных свойств (таких как независимость).

Обучение представлений особенно интересно, потому что дает способ провести обучение без учителя и с частичным привлечением учителя. Часто у нас имеется очень много непомеченных обучающих данных и сравнительно мало помеченных. Обучение с учителем на помеченном подмножестве нередко приводит к сильному переобучению. Обучение с частичным привлечением учителя дает шанс решить эту проблему, поскольку производится и на непомеченных данных тоже. То есть мы можем обучить хорошие представления непомеченных данных, а затем воспользоваться ими для решения задачи обучения с учителем.

Люди и животные умеют учиться на очень небольшом числе помеченных примеров. Мы пока не знаем, как это получается. Объяснить высокую обучаемость человека можно было бы разными причинами – например, мозг может пользоваться очень большими ансамблями классификаторов или техникой байесовского вывода. Популярна гипотеза, согласно которой мозг способен задействовать механизмы обучения без учителя или с частичным привлечением учителя. Есть много способов с пользой употребить непомеченные данные. В этой главе мы сосредоточимся на гипотезе о том, что непомеченные данные можно использовать для обучения хорошего представления.

15.1. Жадное послойное предобучение без учителя

Обучение без учителя сыграло важнейшую роль в воскрешении глубоких нейронных сетей, позволив исследователям впервые обучить глубокую сеть с учителем, не требуя специализации архитектуры, например свертки или рекурсии. Мы называем эту процедуру **предобучением без учителя**, или, если быть точным, **жадным послойным предобучением без учителя**. Это канонический пример того, как представление, обученное для одной задачи (обучение без учителя, пытающееся уловить форму входного распределения), иногда оказывается полезным для другой задачи (обучение с учителем на том же входном домене).

Жадное послойное предобучение без учителя опирается на алгоритм обучения однослойного представления, например: ОМБ, однослойный автокодировщик, модель разреженного кодирования или еще какая-то модель, обучающая латентные представления. Каждый слой предобучается без учителя, получая выход от предыдущего слоя и порождая в качестве выхода новое представление данных с предположительно более простым распределением (или более простыми связями с другими переменными, скажем, категориями, которые нужно предсказать).

Процедуры жадного послойного обучения без учителя давно уже используются, чтобы обойти трудности совместного обучения слоев глубокой нейронной сети с учителем. Этот подход восходит еще к неокогнитрону (Fukushima, 1975). Возрождение глубокого обучения в 2006 году началось с открытия, что процедуру жадного обучения можно использовать для отыскания хороших начальных значений для процедуры совместного обучения всех слоев, причем так можно с успехом обучать даже полносвязные архитектуры (Hinton et al., 2006; Hinton and Salakhutdinov, 2006; Hinton, 2006; Bengio et al., 2007; Ranzato et al., 2007a). До этого открытия считалось практически возможным обучить только глубокие сверточные сети и сети, своей глубиной обязанные рекурсии. Сейчас мы знаем, что для обучения полносвязной глубокой архитектуры необязательно использовать жадное послойное предобучение, но это был первый успешный подход.

Жадное послойное предобучение называется **жадным**, потому что это жадный алгоритм, т. е. он оптимизирует каждую часть решения независимо, а не совместно все части сразу. Оно называется **послойным**, потому что этими независимыми частями являются слои сети. Точнее говоря, в ходе жадного послойного обучения обрабатывается по одному слою за раз – во время обучения k-го слоя все остальные фиксированы. В частности, нижние слои (обучаемые первыми) уже не изменяются после перехода к верхним. В названии процедуры употребляется фраза «**без учителя**», потому что каждый слой обучается с помощью алгоритма обучения представления без учителя. А «**предобучение**» – так как предполагается, что это первый шаг, предшествующий применению алгоритма совместного обучения для настройки работы всех слоев вместе. В контексте задачи обучения с учителем эту процедуру можно рассматривать как регуляризатор (в некоторых экспериментах предобучение уменьшает ошибку тестирования, не уменьшая ошибку обучения) и как вид инициализации параметров.

Обычно слово «предобучение» относится не только к самому этапу предобучения, но и ко всему двухфазному протоколу, в котором фаза предобучения сочетается с фазой обучения с учителем. Фаза обучения с учителем может включать обучение простого классификатора на признаках, выявленных в фазе предобучения, или окончательную настройку с учителем всей сети, построенной в фазе предобучения. В большинстве случаев общая схема обучения одна и та же и не зависит ни от конкретного алгоритма обучения без учителя, ни от типа модели. Хотя выбор алгоритма обучения без учителя, безусловно, влияет на детали, большинство приложений предобучения без учителя следует этому общему протоколу.

Жадное послойное предобучение без учителя можно также использовать для инициализации других алгоритмов обучения без учителя, в частности глубоких автокодировщиков (Hinton and Salakhutdinov, 2006) и вероятностных моделей со многими слоями латентных переменных. К числу таких моделей относятся глубокие сети доверия (Hinton et al., 2006) и глубокие машины Больцмана (Salakhutdinov and Hinton, 2009a). Эти глубокие порождающие модели описаны в главе 20.

Как было сказано в разделе 8.7.4, возможно также жадное послойное предобучение с учителем. В его основе лежит предположение о том, что обучить мелкую сеть проще, чем глубокую, похоже, подтверждающееся в некоторых контекстах (Erhan et al., 2010).

15.1.1. Когда и почему работает предобучение без учителя?

Часто жадное послойное предобучение без учителя может дать значительное уменьшение ошибки тестирования в задачах классификации. Именно из-за этого наблюдения в 2006 году вновь пробудился интерес к глубоким нейронным сетям (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007a). Но во многих других задачах предобучение без учителя либо не дает никакого выигрыша, либо даже наносит заметный вред. В работе Ma et al. (2015) изучалось влияние предобучения на модели предсказания химической активности и обнаружилось, что в среднем предобучение наносит небольшой вред, но в ряде задач оказывается очень полезно. Поскольку предобучение без учителя иногда полезно, но часто вредно, важно понимать, когда и почему оно работает, чтобы определить, применимо ли оно к конкретной задаче.

Алгоритм 15.1. *Протокол жадного послойного предобучения без учителя*
Дано: алгоритм $\mathcal{L}$ обучения признаков без учителя, который принимает обучающий набор примеров и возвращает кодировщик f. Исходные данные представлены в виде матрицы $\boldsymbol{X}$, по одной строке на пример, а $f^{(1)}(\boldsymbol{X})$ – выход первой фазы кодировщика. Если требуется окончательная настройка, то используется обучаемая модель $\mathcal{T}$, которая принимает начальную функцию f, входные примеры $\boldsymbol{X}$ (а в случае если для настройки применяется алгоритм обучения с учителем, – еще и ассоциированные метки $\boldsymbol{Y}$) и возвращает настроенную функцию. Число фаз равно m.

$f \leftarrow$ тождественная функция
$\tilde{\boldsymbol{X}} = \boldsymbol{X}$
for $k = 1, ..., m$ **do**
  $f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}})$
  $f \leftarrow f^{(k)} \circ f$
  $\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}})$
end for
if *окончательная-настройка* **then**
  $f \leftarrow \mathcal{T}(f, \boldsymbol{X}, \boldsymbol{Y})$
end if
Return f

С самого начала уточним, что это обсуждение в основном касается только жадного предобучения без учителя. Существуют другие, принципиально отличающиеся парадигмы обучения с частичным привлечением учителя в применении к нейронным сетям, например виртуальное состязательное обучение, описанное в разделе 7.13. Можно также обучить автокодировщик или порождающую модель одновременно с моделью, обученной с учителем. К таким одношаговым подходам можно отнести дискриминантную ОМБ (Larochelle and Bengio, 2008) и ступенчатую сеть (ladder network) (Rasmus et al., 2015), в которой целевая функция явно представлена в виде суммы двух членов (в одном используются только метки, а в другом – только входные данные).

В предобучении без учителя объединены две идеи. Во-первых, идея о том, что выбор начальных значений параметров глубокой нейронной сети может оказывать существенное регуляризирующее влияние на модель (и, в меньшей степени, способствовать улучшению оптимизации). Во-вторых, общая идея о том, что знание о распределении входных данных может помочь при обучении отображения входов на выходы.

В обоих случаях имеют место сложные и не до конца понятные взаимодействия между несколькими частями алгоритма машинного обучения.

Менее всего понятны причины регуляризирующего эффекта выбора начальных параметров. Когда предобучение только вошло в моду, оно интерпретировалось как инициализация модели таким образом, чтобы она сошлась к одному локальному минимуму, а не к другому. В наши дни локальные минимумы уже не считаются серьезной проблемой для оптимизации нейронной сети. Мы знаем, что стандартные процедуры обучения нейронных сетей обычно не достигают критических точек. Остается возможность, что предобучение инициализирует модель так, что она окажется в точке, которая иначе была бы недоступна, – например, внутри области, окруженной участками, в которых функция стоимости изменяется от примера к примеру так сильно, что мини-пакеты дают лишь очень зашумленную оценку градиента, или участками, где матрица Гессе так плохо обусловлена, что величина шага в методах градиентного спуска должна быть очень малой. Однако у нас нет уверенности в том, какие именно аспекты предобученных параметров сохраняются на этапе обучения с учителем. Это одна из причин, по которой в современных подходах обучение с учителем и без учителя обычно используется одновременно, а не в виде двух этапов, следующих друг за другом. Сложностей, связанных с тем, как в ходе оптимизации на этапе обучения с учителем сохраняется информация, полученная на этапе обучения без учителя, можно также избежать, попросту заморозив параметры экстракторов признаков и используя обучение с учителем только для того, чтобы добавить классификатор поверх уже обученных признаков.

Вторая идея – что алгоритм обучения может использовать информацию, найденную на этапе обучения без учителя, для повышения качества на этапе обучения с учителем, – более понятна. Дело в том, что признаки, полезные в задаче обучения без учителя, могут пригодиться и в задаче обучения с учителем. Например, порождающая модель изображений автомобилей и мотоциклов должна знать о существовании колес и о том, сколько их должно быть. Если нам повезет, то представление колес будет таким, что у модели, обучаемой с учителем, будет к нему удобный доступ. Пока у этого предположения нет теоретического, математически строгого обоснования, поэтому не всегда можно предсказать, какие задачи больше всего выигрывают от такого предобучения без учителя. Многие аспекты этого подхода сильно зависят от того, какая конкретно модель используется. Например, если мы хотим добавить линейный классификатор поверх предобученных признаков, то признаки нужно выбирать так, чтобы классы были линейно разделимы. Подобные свойства часто возникают естественным образом, но это не обязательно. И это еще одна причина, по которой предпочтительно одновременное обучение с учителем и без учителя, – ограничения, налагаемые выходным слоем, естественно включаются с самого начала.

Если рассматривать предобучение без учителя как обучение представления, то можно ожидать, что оно будет более эффективным, когда начальное представление плохое. Яркий пример – погружения слов. Слова, представленные унитарными векторами, не очень информативны, потому что любые два различных унитарных век-

тора находятся на одинаковом расстоянии друг от друга (корень из 2 по норме L^2). Обученные погружения слов естественным образом кодируют сходство между словами по расстоянию между ними. Поэтому предобучение без учителя особенно полезно при обработке слов. Для обработки изображений оно не так полезно, быть может, потому что изображения и так уже принадлежат векторному пространству, в котором расстояние дает низкокачественный показатель сходства.

Если рассматривать предобучение без учителя как регуляризатор, то можно ожидать, что польза от него будет особенно велика, когда число помеченных примеров очень мало. Поскольку источником информации для предобучения без учителя являются непомеченные данные, можно также ожидать, что его качество будет тем лучше, чем больше непомеченных примеров. Преимущества обучения с частичным привлечением учителя в ситуации, когда имеется много непомеченных примеров и мало помеченных и предварительно выполняется обучение без учителя, особенно наглядно проявились в 2011 году, когда методика предобучения без учителя победила в двух международных соревнованиях по переносу обучения (Mesnil et al., 2011; Goodfellow et al., 2011), где число помеченных примеров в задаче варьировалось от нескольких штук до нескольких десятков на каждый класс. Подобные эффекты были также документированы по результатам экспериментов, проведенных в строго контролируемых условиях (Paine et al., 2014).

Возможно, существуют и другие факторы. Например, предобучение без учителя, скорее всего, особенно полезно, когда обучаемая функция очень сложна. Предобучение без учителя отличается от регуляризаторов типа снижения весов тем, что не поощряет обучаемую модель к поиску простой функции, а, скорее, побуждает ее выявлять функции-признаки, полезные для задачи обучения без учителя. Если истинные функции сложны и обусловлены регулярностями входного распределения, то предобучение без учителя может оказаться более подходящим регуляризатором.

Теперь оставим эти рассуждения в стороне и проанализируем некоторые ситуации, когда предобучение без учителя действительно привело к успеху, и объясним, что известно о его причинах. Предобучение без учителя чаще всего применялось для улучшения классификаторов и представляет наибольший интерес с точки зрения уменьшения ошибки на тестовом наборе. Но предобучение без учителя может быть полезно и в задачах, не связанных с классификацией, оно может повысить качество оптимизации, а не просто выступать в роли регуляризатора. Например, оно может уменьшить ошибку реконструкции глубоких автокодировщиков одновременно на обучающих и на тестовых данных (Hinton and Salakhutdinov, 2006).

В работе Erhan et al. (2010) описано много экспериментов для объяснения нескольких успешных случаев применения предобучения без учителя. Уменьшение обеих ошибок – обучения и тестирования – можно объяснить попаданием параметров в область, которая иначе была бы недоступна. Обучение нейронной сети не детерминировано и сходится к новой функции при каждом прогоне. Обучение может остановиться в точке, где градиент оказался мал; в точке, где сработал критерий ранней остановки, предотвращающий переобучение, или в точке, где градиент велик, но трудно определить величину следующего шага из-за стохастичности или плохой обусловленности матрицы Гесса. Нейронные сети, предобученные без учителя, стабильно останавливаются в одной и той же области пространства функций, тогда как сети без предобучения всякий раз останавливаются в новой области. На рис. 15.1 это явление показано наглядно. Область, в которой оказывается предобученная сеть, меньше, и это позво-

ляет предположить, что предобучение уменьшает дисперсию процесса оценивания, что, в свою очередь, снижает риск серьезного переобучения. Иными словами, благодаря предобучению без учителя начальные значения параметров сети оказываются в такой области, которую уже не могут покинуть, так что результаты обучения более стабильны и реже получаются совсем уж никуда не годными.

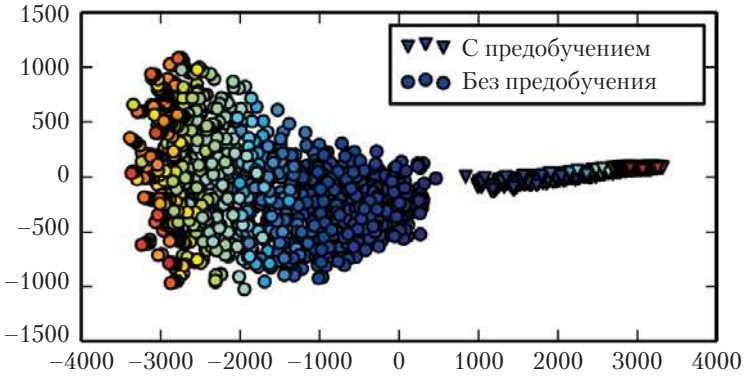

Рис. 15.1 ❖ Визуализация посредством нелинейного проецирования траекторий обучения различных нейронных сетей в *пространстве функций* (не в пространстве параметров, чтобы избежать проблемы отображения нескольких векторов параметров на одну функцию) с различными случайно выбранными начальными значениями, с предобучением и без него. Каждой точки соответствует отдельная нейронная сеть в определенный момент процесса обучения. Рисунок основан на рисунке из работы в Erhan et al. (2010). В пространстве функций координатой является бесконечномерный вектор, ассоциирующий каждый вход $\boldsymbol{x}$ с выходом $\boldsymbol{y}$. В работе Erhan et al. (2010) производилось линейное проецирование на пространство высокой размерности путем конкатенации значений $\boldsymbol{y}$ для многих точек $\boldsymbol{x}$ с последующим нелинейным проецированием на плоскость методом Isomap (Tenenbaum et al., 2000). Цветом представлено время. Все сети инициализированы вблизи центральной точки графика (соответствует области в пространстве функций, где порождаются приблизительно равномерные распределения класса $\boldsymbol{y}$ для большинства входов). С течением времени обучение сдвигает функцию наружу – туда, где предсказания увереннее. Обучение стабильно завершается в одной области, если предобучение применяется, и в другой – если не применяется. Метод Isomap стремится сохранить глобальные относительные расстояния (а стало быть, и объемы), поэтому небольшой размер области, соответствующей предобученным моделям, может указывать на то, что у оценки, полученной с применением предобучения, дисперсия меньше.

В работе Erhan et al. (2010) даны также некоторые ответы на вопрос о том, *когда* предобучение работает лучше всего, – среднее и дисперсия ошибки тестирования уменьшались тем сильнее, чем более глубокая сеть подвергалась предобучению. Следует помнить, что эти эксперименты проводились до того, как были изобретены и обрели популярность современные методы обучения очень глубоких сетей (блоки линейной ректификации, прореживание и пакетная нормировка), поэтому о совместном эффекте предобучения без учителя и современных подходов известно меньше.

Важный вопрос – каким образом предобучение без учителя играет роль регуляризатора? Одна из гипотез состоит в том, что предобучение поощряет алгоритм обучения находить признаки, связанные с истинными причинами порождения наблюдаемых данных. Эта важная идея, положенная в основу многих других алгоритмов, помимо предобучения без учителя, описана более подробно в разделе 15.3.

По сравнению с другими видами обучения без учителя, у предобучения есть недостаток – наличие двух раздельных фаз обучения. Многие стратегии регуляризации позволяют пользователю управлять степенью регуляризации путем изменения единственного гиперпараметра. У предобучения без учителя нет очевидного способа управлять степенью регуляризации, возникающей вследствие этапа обучения без учителя. Вместо этого есть очень много гиперпараметров, эффект которых можно измерить по факту, но зачастую трудно предсказать заранее. Когда вместо предобучения мы производим обучение с учителем и без учителя одновременно, существует единственный гиперпараметр – обычно коэффициент, назначаемый стоимости обучения без учителя, – который определяет, насколько сильно целевая функция без учителя будет регуляризировать модель, обучаемую с учителем. Уменьшение этого коэффициента предсказуемо приводит к ослаблению регуляризации. В случае же предобучения без учителя не существует способа гибко подстраивать степень регуляризации – либо для модели, обучаемой с учителем, берутся предобученные начальные значения параметров, либо нет.

Еще один недостаток двух раздельных фаз обучения состоит в том, что у каждой фазы свои гиперпараметры. Качество второй фазы обычно невозможно предсказать на первой, поэтому между предложением гиперпараметров для первой фазы и их обновлением по результатам второй фазы проходит длительное время. Теоретически самый правильный подход – использовать для выбора гиперпараметров фазы предобучения ошибку на контрольном наборе, как описано в работе Larochelle et al. (2009). Но на практике некоторые гиперпараметры, например число итераций предобучения, удобнее задавать на этапе предобучения, применяя раннюю остановку к целевой функции без учителя, – это хоть и не идеально, но вычислительно обходится куда дешевле, чем использование целевой функции с учителем.

В наши дни от предобучения без учителя отказались почти везде, кроме обработки естественных языков, где естественное представление слов в виде унитарных векторов не несет никакой информации о сходстве и при этом доступны очень большие неразмеченные наборы данных. Преимущество предобучения в этом случае – возможность обучить модель один раз на огромном неразмеченном наборе (скажем, корпусе текстов, содержащем миллиарды слов), найти хорошее представление (обычно слов, но, возможно, и предложений), а затем пользоваться этим представлением или дополнительно настроить его для решения задачи обучения с учителем, в которой обучающий набор содержит гораздо меньше примеров. Впервые этот подход был апробирован в работах Collobert and Weston (2008b), Turian et al. (2010) и Collobert et al. (2011a) и до сих пор широко применяется.

Глубокие сети, основанные на обучении с учителем и регуляризации путем прореживания или пакетной нормировки, во многих задачах не уступают человеку, но только при наличии очень больших размеченных наборов данных. Те же методы превосходят предобучение без учителя на наборах данных среднего размера, например CIFAR-10 и MNIST, насчитывающих примерно по 5000 помеченных примеров на каждый класс. На совсем небольших наборах, таких, например, как данные об альтернативном сплай-

синге, байесовские методы оказываются лучше тех, что основаны на предобучении без учителя (Srivastava, 2013). Потому-то популярность предобучения без учителя и сошла на нет. Тем не менее оно остается важной вехой в истории глубокого обучения и продолжает оказывать влияние на современные подходы. Идея предобучения была обобщена на **предобучение с учителем** (см. раздел 8.7.4) и стала очень распространенным подходом к переносу обучения. Предобучение с учителем в контексте переноса обучения популярно (Oquab et al., 2014; Yosinski et al., 2014) в сверточных сетях, предобученных на наборе данных ImageNet. Для таких обученных сетей публикуются параметры так же, как в задачах обработки естественных языков публикуются предобученные векторы слов (Collobert et al., 2011a; Mikolov et al., 2013a).

15.2. Перенос обучения и адаптация домена

Термины «перенос обучения» и «адаптация домена» относятся к ситуации, когда нечто обученное в одной ситуации (например, распределение P_1) используется для улучшения обобщаемости в другой ситуации (например, при распределении P_2). Это обобщение идеи из предыдущего раздела, в котором мы переносили представление с задачи обучения без учителя на задачу обучения с учителем.

При **переносе обучения** обучаемая модель должна выполнить две или более задач, но предполагается, что многие факторы, объясняющие вариативность P_1, относятся и к изменениям, которые предстоит уловить для обучения P_2. Обычно это интерпретируется в контексте обучения с учителем, когда вход один и тот же, а природа меток может быть разной. Например, мы можем обучиться чему-то, относящемуся к одному набору зрительных категорий, скажем собакам и кошкам, а затем перейти к другим категориям, скажем осам и муравьям. Если в первом случае данных (выбранных из распределения P_1) намного больше, то, возможно, имеет смысл обучить представления, полезные для быстрого обобщения при наличии лишь небольшой выборки из P_2. У многих зрительных категорий есть общие особенности: низкоуровневые признаки – границы и формы, эффекты от применения геометрических преобразований, изменений освещения и т. д. В общем случае перенос обучения, многозадачное обучение (раздел 7.7) и адаптацию домена можно реализовать путем обучения представления, если существуют признаки, полезные в различных ситуациях или задачах, которые соответствуют объясняющим факторам, встречающимся в нескольких ситуациях. Это показано на рис. 7.2, где нижние слои являются общими, а верхние зависят от задачи.

Но иногда общей для разных задач является семантика выхода, а не входа. Например, система распознавания речи должна порождать правильные предложения в выходном слое, но предшествующие слои могут распознавать очень разные варианты одних и тех же фонем или субфонемных огласовок, зависящие от говорящего. В таких случаях разумнее разделять верхние слои нейронной сети и выполнять зависящую от задачи предобработку, как показано на рис. 15.2.

Что касается **адаптации домена**, то задача (оптимальное отображение входа на выход) остается той же самой во всех ситуациях, но распределения входа несколько различаются. Например, рассмотрим задачу анализа эмоциональной окраски, когда нужно определить, выражает ли комментарий положительные или отрицательные эмоции. Комментарии в вебе могут иметь разное происхождение. Проблема адаптации домена возникает, когда предиктор эмоциональной окраски, обученный

на отзывах пользователей книг, видео и музыки, затем используется для анализа комментариев, относящихся, скажем, к потребительской электронике: телевизорам и смартфонам. Можно представить себе, что существует базовая функция, которая выдает эмоциональную окраску – положительную, нейтральную или отрицательную, но, конечно, словарный состав и стиль могут зависеть от предметной области – домена, что затрудняет обобщение с одного домена на другой. Простое предобучение без учителя (с помощью шумоподавляющих автокодировщиков) показало прекрасные результаты при анализе эмоциональной окраски посредством адаптации домена (Glorot et al., 2011b).

Рис. 15.2 ❖ Пример архитектуры для многозадачного обучения и переноса обучения, когда у переменной **y** общая для нескольких задач семантика, а вход **x** имеет разную семантику (и, возможно, даже разную размерность) для каждой задачи (или, к примеру, каждого пользователя): $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$. Нижние уровни (до переключателя) зависят от задачи, а верхние являются общими. Нижние уровни обучаются транслировать зависящие от задачи входы в универсальный набор признаков

Родственная задача – **дрейф концепций** (concept drift), ее можно рассматривать как форму переноса обучения в силу постепенных изменений распределения данных со временем. И дрейф концепций, и перенос обучения можно считать особыми видами многозадачного обучения. Хотя термин «многозадачное обучение» чаще применяется к задачам обучения с учителем, более общее понятие переноса обучения применимо также к обучению без учителя и к обучению с подкреплением.

Во всех этих случаях цель – воспользоваться знаниями, приобретенными в первой конфигурации, для извлечения информации, которая может пригодиться для обучения или даже прямого предсказания во второй конфигурации. Ключевая идея обучения представлений состоит в том, что одно и то же представление может быть полезно в обеих конфигурациях. А это позволяет обогатить представление, пользуясь обучающими данными, доступными для обеих задач.

Как уже упоминалось, глубокое обучение без учителя в применении к переносу обучения снискало успех в нескольких соревнованиях по машинному обучению (Mesnil et al., 2011; Goodfellow et al., 2011). В первом из них предлагалась такая задача.

Каждому участнику сначала был выдан набор данных из первой конфигурации (выборка из распределения P_1) с примерами, относящимися к какому-то множеству категорий. На нем участники должны были обучить хорошее пространство признаков (отображение исходных данных на некоторое представление) – такое, что применение этого обученного преобразования к входным данным для переноса обучения (выборке из распределения P_2) позволяет на небольшом количестве помеченных примеров обучить линейный классификатор, который бы хорошо обобщался. Среди самых поразительных результатов, полученных в ходе этого соревнования, был тот факт, что по мере использования все более глубоких представлений (обученных без всякого присутствия учителя на данных, выбранных из первого распределения P_1) кривая обучения на новых категориях второй конфигурации P_2 (перенос) оказывается гораздо лучше. При наличии глубокого представления для достижения асимптотической обобщаемости на перенесенных задачах нужно меньше помеченных примеров.

Два крайних случая переноса обучения – **обучение на одном примере** (one-shot learning) и **обучение без примеров** (zero-shot learning), или **обучение без данных** (zero-data learning). В первом случае для перенесенной задачи имеется только один помеченный пример, а во втором – ни одного.

Обучение на одном примере (Fei-Fei et al., 2006) возможно, потому что представление обучается четко различать классы на первом этапе. А на этапе переноса обучения достаточно одного помеченного примера, чтобы вывести метку многих возможных тестовых примеров, сконцентрированных вокруг одной точки в пространстве представления. Это работает, если факторы вариативности, соответствующие этим инвариантам, четко отделены от других факторов в пространстве обученного представления и если мы каким-то образом сумели установить, какие факторы существенны, а какие несущественны для различения объектов определенных категорий.

Обучение без примеров можно проиллюстрировать на задаче, в которой обучаемая модель должна прочитать большой корпус текстов, а затем решить некую задачу распознавания объектов. Распознать класс объекта можно, даже никогда не видя его изображения, если у класса имеется достаточно хорошее текстовое описание. Например, прочитав, что у кошки четыре ноги и заостренные уши, обучаемая система могла бы предположить, что на картинке изображена кошка, хотя никогда ее не видела.

Обучение без данных (Larochelle et al., 2008) и обучение без примеров (Palatucci et al., 2009; Socher et al., 2013b) возможны только потому, что во время обучения использовалась дополнительная информация. Можно считать, что в обучении без данных присутствуют три случайные величины: традиционный вход $\boldsymbol{x}$, традиционный выход или метки $\boldsymbol{y}$ и дополнительная случайная величина, описывающая задачу, T. Модель обучается оценивать условное распределение $p(\boldsymbol{y} \mid \boldsymbol{x}, T)$, где T – описание задачи, которую должна решать модель. В нашем примере распознавания кошек по их текстовому описанию выходом является бинарная переменная y – такая, что $y = 1$ означает «да», а $y = 0$ – «нет». Тогда величина T представляет вопросы, требующие ответа, например: «Присутствует ли в изображении кошка?» Если имеется обучающий набор, содержащий непомеченные примеры объектов, принадлежащих тому же пространству, что и T, то мы могли бы вывести смысл экземпляров T, которых раньше не видели. В нашем случае важно наличие непомеченных текстовых данных, содержащих предложения вида «у кошки четыре ноги» или «у кошки заостренные уши».

Для обучения без примеров требуется, чтобы T была представлена в виде, допускающем какое-то обобщение. Например, T не может быть просто унитарным кодом,

обозначающим категорию объекта. В работе Socher et al. (2013b) предложено альтернативное распределенное представление категорий объектов посредством обученного погружения слов, ассоциированных с каждой категорией.

Похожий феномен встречается в машинном переводе (Klementiev et al., 2012; Mikolov et al., 2013b; Gouws et al., 2014): имеются слова на одном языке и связи между словами, которые можно обучить на корпусе одноязычных текстов; с другой стороны, имеются переведенные предложения, сопоставляющие слова на двух разных языках. И хотя у нас может не быть помеченных примеров перевода слова A на языке X в слово B на языке Y, мы можем произвести обобщение и предложить перевод слова A, поскольку обучили распределенные представления слов языка X и языка Y, а затем создали связь (возможно, двустороннюю) между обоими пространствами посредством обучающих примеров, состоящих из соответственных пар предложений на двух языках. Такой перенос будет особенно успешным, если все три составные части (два представления и связи между ними) обучались совместно.

Обучение без примеров – частный случай переноса обучения. Тот же принцип объясняет, как можно провести **многомодальное обучение**, которое строит представление в двух разных модальностях и связь (в общем случае совместное распределение) между парами $(\boldsymbol{x}, \boldsymbol{y})$, где $\boldsymbol{x}$ – наблюдение в одной модальности, а $\boldsymbol{y}$ – в другой (Srivastava and Salakhutdinov, 2012). Путем обучения всех трех наборов параметров (из $\boldsymbol{x}$ в его представление, из $\boldsymbol{y}$ в его представление и связь между обоими представлениями) концепции из одного представления привязываются к концепциям из другого, и наоборот, что позволяет осмысленно производить обобщение на новые пары. Эта процедура показана на рис. 15.3.

15.3. Разделение каузальных факторов с частичным привлечением учителя

В связи с обучением представлений возникает важный вопрос: из-за чего одно представление лучше другого? Одна из гипотез заключается в том, что идеальным является представление, в котором признаки соответствуют объясняющим причинам наблюдаемых данных, причем разные признаки или направления в пространстве признаков соответствуют разным причинам, так что представление отделяет причины друг от друга. Следуя этой гипотезе, мы сначала ищем хорошее представление для $p(\boldsymbol{x})$. Оно может также оказаться хорошим представлением для вычисления $p(\boldsymbol{y} \mid \boldsymbol{x})$, если $\boldsymbol{y}$ – одна из наиболее выраженных причин $\boldsymbol{x}$. Эта идея вдохновляла немало исследований по глубокому обучению аж с 1990-х годов (Becker and Hinton, 1992; Hinton and Sejnowski, 1999). Другие соображения о том, когда обучение с частичным привлечением учителя может превосходить чистое обучение с учителем, см. в разделе 1.2 работы Chapelle et al. (2006).

В других подходах к обучению представлений нас часто интересовало представление, которое проще моделировать, – например, с разреженными или независимыми друг от друга элементами. Представление, которое четко разделяет каузальные факторы, необязательно просто моделируется. Однако у гипотезы, выдвигаемой в обоснование обучения с частичным привлечением учителя посредством представления без учителя, есть вторая часть – для многих задач ИИ справедливо следующее утверждение: если мы можем получить объяснение наблюдаемым фактам, то обычно

легко разделить отдельные атрибуты. Точнее говоря, если представление $\boldsymbol{h}$ представляет многие объясняющие причины наблюдения $\boldsymbol{x}$, а выходы $\boldsymbol{y}$ входят в число самых выраженных причин, то легко предсказать $\boldsymbol{y}$ по $\boldsymbol{h}$.

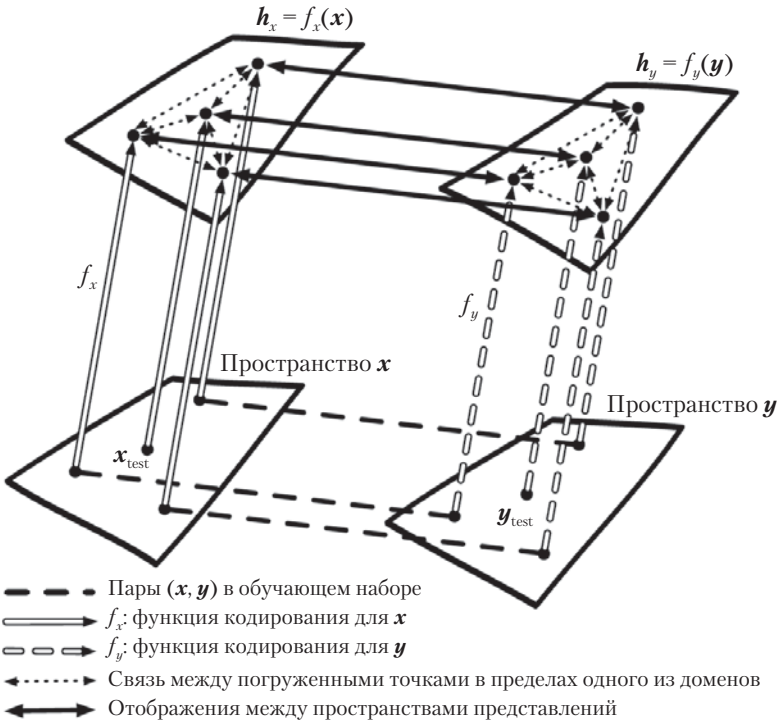

Рис. 15.3 ❖ Перенос обучения между доменами $\boldsymbol{x}$ и $\boldsymbol{y}$ делает возможным обучение без примеров. Помеченные или непомеченные примеры $\boldsymbol{x}$ позволяют обучить функцию представления f_x и аналогично с примерами $\boldsymbol{y}$ и функцией f_y. Каждое применение функций f_x и f_y обозначено направленной вверх стрелкой, а стиль стрелок показывает, какая функция применена. Расстояние в пространстве $\boldsymbol{h}_x$ определяет меру сходства между любой парой точек в пространстве $\boldsymbol{x}$, возможно, более осмысленное, чем расстояние в самом пространстве $\boldsymbol{x}$. Аналогично расстояние в пространстве $\boldsymbol{h}_y$ определяет меру сходства между парами точек в пространстве $\boldsymbol{y}$. Обе эти функции сходства обозначаются штриховыми двусторонними стрелками. Помеченные примеры (штриховые горизонтальные линии) – это пары $(\boldsymbol{x}, \boldsymbol{y})$, которые позволяют обучить одностороннее или двустороннее отображение (сплошная двусторонняя стрелка) между представлениями $f_x(\boldsymbol{x})$ и $f_y(\boldsymbol{y})$ и скрепить эти представления друг с другом. После этого обучение без примеров производится следующим образом. Мы можем ассоциировать изображение $\boldsymbol{x}_{\text{test}}$ со словом $\boldsymbol{y}_{\text{test}}$, даже если нет ни одного изображения этого слова, просто потому что представления слова $f_y(\boldsymbol{y}_{\text{test}})$ и изображения $f_x(\boldsymbol{x}_{\text{test}})$ можно связать между собой с помощью отображения между пространствами представлений. Это работает, потому что, хотя изображение и слово никогда не были объединены в пару, соответствующие им векторы признаков $f_x(\boldsymbol{x}_{\text{test}})$ и $f_y(\boldsymbol{y}_{\text{test}})$ сопоставлены друг с другом. Идея рисунка подсказана Грантом Хачатряном

Сначала поговорим о том, как обучение с частичным привлечением учителя может закончиться неудачей, поскольку обучение $p(\mathbf{x})$ без учителя ничем не помогает обучению $p(\mathbf{y} \mid \mathbf{x})$. Рассмотрим, к примеру, случай, когда $p(\mathbf{x})$ имеет равномерное распределение и мы хотим обучить $f(\boldsymbol{x}) = \mathbb{E}[\mathbf{y} \mid \boldsymbol{x}]$. Очевидно, что наблюдение одного лишь обучающего набора значений $\boldsymbol{x}$ не дает никакой информации о $p(\mathbf{y} \mid \mathbf{x})$.

Теперь рассмотрим простой пример ситуации, когда обучение с частичным привлечением учителя может принести успех. Пусть распределение $\mathbf{x}$ – это смесь распределений, в которой каждой компоненте соответствует одно значение $\mathbf{y}$, как показано на рис. 15.4. Если компоненты смеси четко разделены, то моделирование $p(\mathbf{x})$ точно определит, где какая компонента, и тогда одного помеченного примера каждого класса будет достаточно для идеального обучения $p(\mathbf{y} \mid \mathbf{x})$. Но в более общем случае что может связывать $p(\mathbf{y} \mid \mathbf{x})$ и $p(\mathbf{x})$?

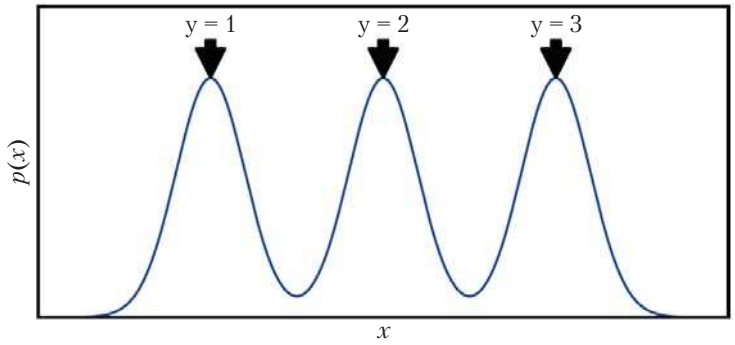

Рис. 15.4 ❖ Смесовая модель. Плотность распределения x является смесью трех компонент. Каждая компонента соответствует истинному объясняющему фактору y. Поскольку компоненты смеси (например, классы естественных объектов в изображениях) статистически выражены, то простое моделирование $p(x)$ без учителя, при полном отсутствии помеченных примеров, уже выявляет фактор y

Если величина $\mathbf{y}$ прочно ассоциируется с одним из каузальных факторов $\mathbf{x}$, то $p(\mathbf{x})$ и $p(\mathbf{y} \mid \mathbf{x})$ будут сильно связаны, и обучение без учителя представления, которое пытается разделить каузальные факторы вариативности, вероятно, будет полезно в качестве стратегии обучения с частичным привлечением учителя.

Предположим, что $\mathbf{y}$ – один из каузальных факторов $\mathbf{x}$, и пусть $\mathbf{h}$ – представление всех таких факторов. Можно считать, что истинный порождающий процесс структурирован в виде следующей ориентированной графической модели, в которой $\mathbf{h}$ – родитель $\mathbf{x}$:

$$p(\mathbf{h}, \mathbf{x}) = p(\mathbf{x} \mid \mathbf{h})p(\mathbf{h}). \tag{15.1}$$

Следовательно, безусловная вероятность данных равна

$$p(\boldsymbol{x}) = \mathbb{E}_{\mathbf{h}} p(\boldsymbol{x} \mid \boldsymbol{h}). \tag{15.2}$$

Из этого простого наблюдения мы заключаем, что наилучшей моделью $\mathbf{x}$ (с точки зрения обобщаемости) будет такая, которая вскрывает приведенную выше «истинную» структуру, в которой $\boldsymbol{h}$ – латентная переменная, объясняющая наблюдаемую

вариативность $\boldsymbol{x}$. Поэтому в ходе обучения «идеального» представления следует выявить все такие латентные факторы. Если $\mathbf{y}$ – один из них (или тесно связан с одним из них), то будет легко научиться предсказывать $\mathbf{y}$ по такому представлению. Мы также видим, что условное распределение $\mathbf{y}$ при условии $\mathbf{x}$ по правилу Байеса связано с компонентами в уравнении выше:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{y}) p(\mathbf{y})}{p(\mathbf{x})}. \tag{15.3}$$

Таким образом, безусловная вероятность $p(\mathbf{x})$ тесно связана с условной вероятностью $p(\mathbf{y} \mid \mathbf{x})$, и знание структуры первой должно помочь при обучении последней. Следовательно, в ситуациях, отвечающих сформулированным предположениям, обучение с частичным привлечением учителя должно повысить качество.

Важная исследовательская проблема связана с тем фактом, что у большинства наблюдений чрезвычайно много объясняющих причин. Допустим, что $\mathbf{y} = \mathrm{h}_i$, но обучаемая без учителя модель не знает, какой именно фактор h_i. Решение в лоб – обучить без учителя представление, которое улавливает *все* сколько-нибудь выраженные порождающие факторы h_j и отделяет их друг от друга, так что становится легко предсказать $\mathbf{y}$ по $\mathbf{h}$ вне зависимости от того, какой h_i ассоциирован с $\mathbf{y}$.

На практике такое лобовое решение не годится, потому что невозможно выделить все или хотя бы большую часть факторов вариативности, оказывающих влияние на наблюдение. Например, если имеется визуальная сцена, то должно ли представление кодировать самые мельчайшие объекты на заднем плане? Хорошо известен психологический феномен – человек не воспринимает изменений в окружающей среде, если они не относятся непосредственно к тому делу, которым он занимается, – см., например, Simons and Levin (1998). На переднем крае обучения с частичным привлечением учителя находится вопрос о том, что же кодировать в конкретной ситуации. В настоящее время есть две основные стратегии обращения с большим числом обусловливающих причин: одновременно использовать сигналы обучения с учителем и без учителя, так чтобы модель выбрала наиболее релевантные факторы вариативности, или брать гораздо более объемные представления в режим чистого обучения без учителя.

Постепенно вырисовывается стратегия обучения без учителя – изменить определение того, какие объясняющие причины являются самыми выраженными. Исторически автокодировщики и порождающие модели обучались оптимизировать фиксированный критерий, зачастую похожий на среднеквадратическую ошибку. Именно такие критерии и определяют, что считать выраженными причинами. Например, среднеквадратическая ошибка в применении к пикселям изображения неявно означает, что объясняющая причина является выраженной, только если она сильно изменяет яркость большого числа пикселей. Это может стать проблемой, если решаемая задача подразумевает взаимодействие с малыми объектами. На рис. 15.5 приведен пример робототехнической задачи, в которой автокодировщик не смог научиться кодировать мячик для игры в настольный теннис. Но тот же самый робот успешно взаимодействует с более крупными объектами, например бейсбольными мячами, которые более выражены в терминах среднеквадратической ошибки.

Возможны и другие определения выраженности. Например, если группа пикселей образует легко распознаваемый паттерн, пусть даже не экстремально яркий или темный, то такой паттерн можно было бы счесть чрезвычайно выраженным. Чтобы реализовать такое определение выраженности, можно воспользоваться недавно появив-

шимся подходом – **порождающими состязательными сетями** (generative adversarial network) (Goodfellow et al., 2014c). В нем порождающая модель обучается обманывать классификатор с прямым распространением, который пытается распознавать все примеры, порожденные моделью, как фальшивки, а все примеры из обучающего набора – как настоящие. В этой системе любой структурный паттерн, который сеть прямого распространения может распознать, является сильно выраженным.

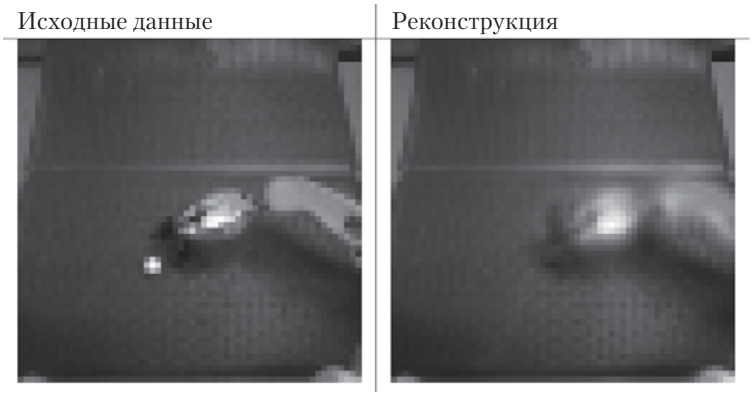

Рис. 15.5 ❖ Автокодировщик, обученный с применением среднеквадратической ошибки, не справился с робототехнической задачей реконструкции мячика для настольного тенниса. Само присутствие мячика и все его пространственные координаты – важные каузальные факторы, имеющие отношение к задаче робота. К сожалению, емкость автокодировщика ограничена, и обучение с использованием среднеквадратической ошибки не позволило идентифицировать мячик как достаточно выраженный фактор для кодирования. Изображения любезно предоставила Челси Финн

Порождающая состязательная сеть более подробно описана в разделе 20.10.4. Сейчас же нам достаточно понимать, что сети обучаются тому, что считать выраженным. В работе Lotter et al. (2015) показано, что модели, обученные генерировать изображения головы человека, часто не генерируют уши, если обучались с применением среднеквадратической ошибки, но благополучно делают это, если при обучении использовалась состязательная сеть. Поскольку уши не являются слишком яркими или темными на фоне окружающей кожи, то они не считаются выраженным фактором относительно среднеквадратической ошибки в качестве функции потерь, но уши имеют характерную форму и находятся в одном и том же месте, поэтому сеть прямого распространения можно легко обучить их распознаванию, а значит, в модели на основе порождающей состязательной сети они будут ярко выраженными факторами. Примеры изображений см. на рис. 15.6. Порождающие состязательные – лишь один шаг на пути к определению того, какие факторы следует представлять. Мы ожидаем, что будущие исследователи найдут более удачные способы определения того, какие факторы представлять, и разработают механизмы представления различных факторов, зависящие от задачи.

В работе Schölkopf et al. (2012) отмечено, что преимущество обучения каузальных факторов состоит в том, что если для истинного порождающего процесса $\mathbf{x}$ – следствие, а $\mathbf{y}$ – причина, то моделирование $p(\mathbf{x} \mid \mathbf{y})$ устойчиво относительно изменения

$p(\mathbf{y})$. Если бы мы изменили причинно-следственную связь на противоположную, то это уже было бы неверно, потому что согласно правилу Байеса $p(\mathbf{x} \mid \mathbf{y})$ была бы чувствительна к изменениям $p(\mathbf{y})$. Очень часто, когда мы рассматриваем изменения в распределении из-за различных доменов, нестационарности во времени или изменений в характере задачи, *каузальные механизмы остаются инвариантными* («законы Вселенной постоянны»), тогда как маргинальное распределение причин вариативности может изменяться. Поэтому лучшей обобщаемости и устойчивости к разного рода изменениям можно ожидать, когда обученная порождающая модель стремится выявить каузальные факторы $\mathbf{h}$ и $p(\mathbf{x} \mid \mathbf{h})$.

Рис. 15.6 ❖ Предсказательные порождающие сети – пример, демонстрирующий, как важно обучиться тому, какие признаки являются выраженными. В данном случае порождающая сеть обучена предсказывать внешний вид трехмерной модели головы человека при заданном угле обзора. (*Слева*) Истинная картина. Это то изображение, которое должна выдать сеть. (*В центре*) Изображение, порожденное сетью, обученной с применением одной лишь среднеквадратической ошибки. Поскольку уши не слишком выделяются по яркости на фоне соседних участков кожи, модель не сочла эти признаки достаточно выраженными для включения в представление. (*Справа*) Изображение, порожденное моделью, которая была обучена с применением комбинации среднеквадратической ошибки и состязательной потери. При такой функции стоимости уши – достаточно выраженный признак, потому что образуют предсказуемый паттерн. Выявление в процессе обучения тех объясняющих причин, которые достаточно важны и релевантны модели, – важное направление современных исследований. Рисунки взяты из работы Lotter et al. (2015)

15.4. Распределенное представление

Распределенные представления концепций, т. е. представления, составленные из нескольких элементов, которые можно задавать независимо друг от друга, – один из самых важных инструментов обучения представлений. Связано это с тем, что распределенное представление дает возможность использовать n признаков с k значениями для описания k^n различных концепций. Как мы уже не раз видели в этой книге, и в нейронных сетях с несколькими скрытыми блоками, и в вероятностных моделях с несколькими латентными переменными используется стратегия распределенных представлений. Теперь добавим еще одно наблюдение. В основе многих алгоритмов

глубокого обучения лежит предположение, что скрытые блоки можно обучить представлениям каузальных факторов, объясняющих данные, как описано в разделе 15.3. При таком подходе распределенные представления возникают естественно, потому что каждому направлению в пространстве представления можно поставить в соответствие значение отдельной обусловливающей конфигурационной переменной.

Примером распределенного представления является вектор n бинарных признаков, принимающий 2^n значений, каждому из которых может соответствовать отдельная область пространства входов, как показано на рис. 15.7. Это можно сравнить с *символическим представлением*, в котором вход ассоциирован с одним символом или категорией. Если словарь содержит n символов, то можно представить себе n детекторов признаков, каждый из которых определяет присутствие ассоциированной категории. В этом случае возможно только n различных конфигураций пространства представления, вырезающих n областей в пространстве входов, как показано на рис. 15.8. Такое символическое представление называется унитарным, поскольку описывается бинарным вектором, в котором единичным может быть только один из n бит. Символическое представление – частный случай более широкого класса нераспределенных представлений, которые могут состоять из многих элементов, но не допускают осмысленного контроля над каждым из них.

Ниже приведено несколько примеров алгоритмов обучения, основанных на нераспределенных представлениях.

- Методы кластеризации, включая алгоритм k средних: каждая входная точка относится ровно к одному кластеру.
- Алгоритмы k ближайших соседей: с заданным входом ассоциируется один или несколько примеров-прототипов, или шаблонов. Если $k > 1$, то каждый вход описывается несколькими значениями, но ими нельзя управлять независимо, поэтому по-настоящему распределенным такое представление назвать нельзя.
- Решающие деревья: для каждого входа активируется только один листовый узел (и все узлы на пути к нему от корня).
- Гауссовы смеси и коллективы экспертов: шаблоны (центры кластеров), или эксперты теперь ассоциируются со *степенью активации*. Как и в случае алгоритма k ближайших соседей, каждый вход представляется несколькими значениями, но управлять ими по отдельности невозможно.
- Ядерные методы с гауссовым (или другим локальным) ядром: хотя степень активации каждого «опорного вектора», или примера-шаблона теперь непрерывна, возникают те же проблемы, что с гауссовыми смесями.
- Модели языка или перевода, основанные на n-граммах: множество контекстов (последовательностей символов) разбивается в соответствии со структурой суффиксного дерева. Листовый узел может соответствовать, например, контексту, в котором два последних слова равны w_1 и w_2. Для каждого листового узла вычисляется своя оценка параметров (при этом возможно некоторое разделение параметров).

Для некоторых из описанных нераспределенных алгоритмов выход для каждой части не постоянный, а является интерполяцией выходов соседних областей. Связь между числом параметров (или примеров) и числом определяемых ими областей остается линейной.

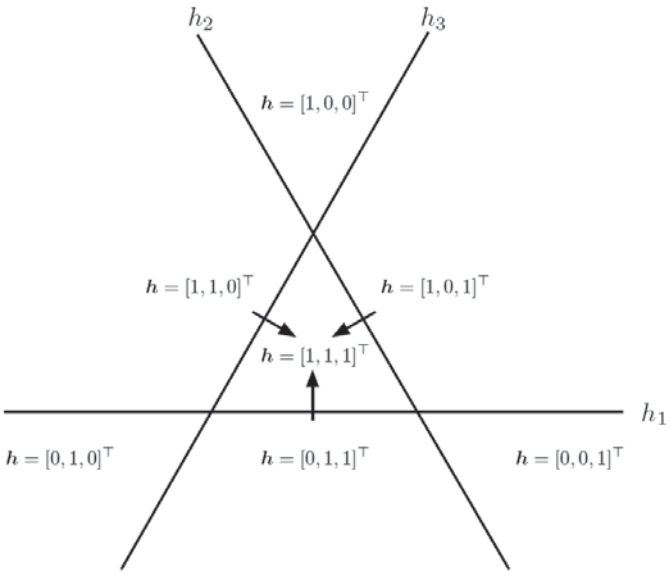

Рис. 15.7 ❖ Как алгоритм, основанный на распределенном представлении, разбивает пространство входов на области. В данном случае имеются три бинарных признака h_1, h_2 и h_3. Каждый признак определен посредством бинаризации выхода обученного линейного преобразования и разбивает $\mathbb{R}^2$ на две полуплоскости. Обозначим h_i^+ множество входных точек, для которых $h_i = 1$, а h_i^- – множество точек, для которых $h_i = 0$. На рисунке каждая прямая представляет решающую границу для одного h_i, а соответствующая стрелка направлена в сторону h_i^+ от границы. Представление в целом принимает уникальное значение в каждой области, на которые разбивают плоскость три полуплоскости. Например, значение представления $[1, 1, 1]^\top$ соответствует области $h_1^+ \cap h_2^+ \cap h_3^+$. Сравните это с нераспределенными представлениями на рис. 15.8. В общем случае d-мерного входа распределенное представление разбивает $\mathbb{R}^d$ на области пересечения полупространств, а не полуплоскостей. Распределенное представление с n признаками назначает уникальные коды $O(n^d)$ различным областям, тогда как алгоритм ближайших соседей с n примерами – только n областям. Поэтому распределенное представление способно различить экспоненциально больше областей, чем нераспределенное. Следует иметь в виду, что не все значения $\boldsymbol{h}$ практически реализуемы (в данном примере нет значения $\boldsymbol{h} = \mathbf{0}$) и что линейный классификатор поверх распределенного представления не может назначить различные классы всем соседним областям; даже у глубокой сети с линейными порогами VC-размерность имеет порядок только $O(w \log w)$, где w – количество весов (Sontag, 1998). Комбинация мощного слоя представления со слабым слоем классификации может оказаться хорошим регуляризатором; классификатору, пытающемуся обучиться различению концепций «человек» и «не человек», не нужно назначать разные классы входам, представленным как «женщина в очках» и «мужчина без очков». Благодаря этому ограничению на емкость каждый классификатор фокусируется на небольшом числе h_i, а $\boldsymbol{h}$ обучается представлять классы линейно разделимым способом

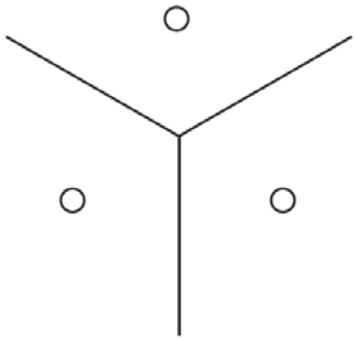

Рис. 15.8 ❖ Как алгоритм ближайшего соседа разбивает пространство входов на различные области. Этот пример алгоритма обучения, основанного на нераспределенном представлении. У нераспределенных алгоритмов может быть разная геометрия, но все они обычно разбивают пространство входов на несколько областей, *в каждой из которых свой набор параметров*. Преимущество нераспределенного подхода состоит в том, что при наличии достаточного числа параметров можно аппроксимировать обучающий набор, не решая трудных уравнений оптимизации, поскольку можно *независимо* выбирать различные выходы для каждой области. Недостаток же в том, что такие нераспределенные модели обобщаются только локально, исходя из априорного предположения о гладкости, поэтому трудно обучить сложную функцию, для которой число пиков и впадин превышает располагаемое число примеров. Сравните с распределенным представлением на рис. 15.7

С различием между распределенным и символическим представлениями связана еще одна важная идея: *возможность обобщения проистекает из разделения атрибутов* между разными концепциями. Как чистые символы слова «кошка» и «собака» так же далеки друг от друга, как два любых других символа. Но если ассоциировать их с осмысленным распределенным представлением, то многое из того, что можно сказать о кошках, обобщается на собак – и наоборот. Например, в распределенном представлении могут быть атрибуты «имеет_мех» и «число_ног», и их значения одинаковы для погружений слов «кошка» и «собака». Нейронные языковые модели, работающие с распределенными представлениями слов, обобщаются гораздо лучше моделей, работающих напрямую с унитарными представлениями слов (см. раздел 2.4). Распределенные представления индуцируют полезное метрическое пространство, в котором расстояние между семантическими близкими концепциями (или входами) мало, – этим свойством чисто символические представления не обладают.

Когда и почему использование распределенного представления в качестве части алгоритма обучения может дать статистическое преимущество? Когда сложную на первый взгляд структуру можно компактно представить с помощью небольшого числа параметров. Некоторые традиционные нераспределенные алгоритмы обучения обобщаются только в предположении гладкости, согласно которому если $u \approx v$, то целевая функция f, которую предстоит обучить, такова, что $f(u) \approx f(v)$. Есть много способов формализовать это предположение, но конечный результат всегда один: если имеется пример (x, y), для которого известно, что $f(x) \approx y$, то мы выбираем оценку

$\hat{f}$, которая приблизительно удовлетворяет этим ограничениям и при этом как можно меньше изменяется при переходе к близкому входу $x + \varepsilon$. Очевидно, что это предположение очень полезно, но оно подвержено проклятию размерности: чтобы обучить целевую функцию, которая многократно возрастает и убывает во многих областях[1], число примеров должно быть никак не меньше числа различимых областей. Можно считать каждую такую область категорией или символом: если у каждого символа (или области) имеется отдельная степень свободы, то мы можем обучить произвольный декодер, отображающий символ в значение. Однако такой декодер не обобщается на новые символы для новых областей.

Если нам повезет, то у целевой функции может оказаться еще какая-то регулярность, помимо гладкости. Например, сверточная сеть с max-пулингом способна распознать объект вне зависимости от его положения в изображении, пусть даже параллельный перенос в пространстве не соответствует гладкому преобразованию пространства входов.

Рассмотрим частный случай алгоритма обучения распределенного представления, который извлекает бинарные признаки посредством бинаризации линейных функций входа. Каждый бинарный признак в таком представлении разбивает $\mathbb{R}^d$ на два полупространства, как показано на рис. 15.7. Экспоненциально большое число областей, высекаемых n полупространствами, определяет, сколько областей способно различить такое распределенное представление. Сколько же именно областей генерируется конфигурацией n гиперплоскостей в $\mathbb{R}^d$? Применяя общий результат о пересечении гиперплоскостей (Zaslavsky, 1975), можно показать (Pascanu et al., 2014b), что число областей, различимых таким представлением бинарных признаков, равно

$$\sum_{j=0}^{d} \binom{n}{j} = O(n^d). \tag{15.4}$$

Таким образом, мы видим, что число областей экспоненциально зависит от размера входа и полиномиально от числа скрытых блоков.

Тем самым мы получаем геометрическое объяснение обобщаемости распределенного представления: имея $O(nd)$ параметров (n линейных пороговых признаков в $\mathbb{R}^d$), мы можем представить $O(n^d)$ различных областей в пространстве входов. Если бы мы не делали никаких предположений о данных, использовали представление с одним уникальным символом для каждой области и отдельные параметры для каждого символа для распознавания соответствующей ему области $\mathbb{R}^d$, то для задания $O(n^d)$ областей потребовалось бы $O(n^d)$ примеров. Вообще, аргументацию в пользу распределенного представления можно обобщить на случай, когда вместо линейных пороговых блоков используются экстракторы нелинейных, возможно, непрерывных признаков для каждого атрибута распределенного представления. В этом случае аргументация сводится к тому, что если параметрическое преобразование с k параметрами может обучиться распознавать r областей в пространстве входов, где $k \ll r$, и если такое пред-

---

[1] Теоретически может потребоваться обучить функцию, поведение которой различается в экспоненциально большом числе областей: в d-мерном пространстве, где нужно различать, по крайней мере, два значения по каждому измерению, нам может понадобиться функция f, которая принимает разные значения в 2^d областях, для ее обучения необходимо $O(2^d)$ примеров.

ставление оказалось полезно для интересующей нас задачи, то, возможно, мы могли бы добиться гораздо лучшей обобщаемости по сравнению с нераспределенным представлением, для которого нужно было бы иметь $O(r)$ примеров, чтобы получить те же признаки и ассоциированное с ними разбиение пространства входов на r областей. Коль скоро для представления модели требуется меньше примеров, то и подстраиваться придется под меньшее число параметров, а значит, для хорошей обобщаемости понадобится гораздо меньше примеров.

Дополнительный аргумент, объясняющий хорошую обобщаемость моделей на основе распределенных представлений, состоит в том, что их емкость остается ограниченной, несмотря на способность кодировать так много различных областей. Например, VC-размерность нейронной сети линейных пороговых блоков равна всего $O(w \log w)$, где w – количество весов (Sontag, 1998). Это ограничение возникает, потому что хоть мы и можем назначить очень много уникальных кодов пространству представления, но мы не можем ни использовать абсолютно все пространство кодов, ни обучить произвольные функции, отображающие пространство представления $\boldsymbol{h}$ на выходы $\boldsymbol{y}$ с применением линейного классификатора. Поэтому использование распределенного представления в сочетании с линейным классификатором выражает априорное предположение о том, что подлежащие распознаванию классы линейно разделимы как функция каузальных факторов, запомненных в $\boldsymbol{h}$. Обычно мы хотим обучить категории, например множество всех изображений зеленых объектов или всех изображений автомобилей, но не категории, требующие нелинейной XOR-логики. Например, нам, как правило, не требуется разбивать данные на класс красных легковых автомобилей плюс зеленых грузовиков и класс зеленых легковых автомобилей плюс красных грузовиков.

Обсуждавшиеся выше идеи были довольно абстрактными, но их можно проверить экспериментально. В работе Zhou et al. (2015) показано, что скрытые блоки глубокой сверточной сети, обученной на эталонных наборах данных ImageNet и Places, обучаются признакам, которые часто можно интерпретировать как метки, которые присвоил бы человек. Конечно, на практике не всегда бывает так, что для признаков, которым обучились скрытые блоки, в естественном языке имеется простое название, но интересно наблюдать, как они вырисовываются ближе к верхним уровням лучших глубоких сетей, применяемых в компьютерном зрении. У таких признаков есть одна общая черта: можно представить себе, что *каждому из них можно обучиться, не видя всех конфигураций остальных*. В работе Radford et al. (2015) показано, что порождающая модель может обучить представление изображений лица, так что различные направления в пространстве представления будут улавливать разные факторы вариативности. На рис. 15.9 показано, что одно направление в пространстве представления соответствует полу человека, а другое – наличию очков. Эти признаки были выявлены автоматически, а не заданы заранее. Классификаторам в скрытых блоках не нужны метки: метод градиентного спуска для заданной целевой функции естественным образом обучает сеть семантически значимым признакам, если такие признаки нужны в задаче. Мы можем обучить сеть различию между мужчиной и женщиной или наличию или отсутствию очков, не пытаясь охарактеризовать все конфигурации $n - 1$ прочих признаков с помощью примеров, покрывающих все комбинации их значений. Такая форма статистической разделимости и есть то, что открывает возможность обобщения на новые конфигурации признаков человека, которые не предъявлялись во время обучения.

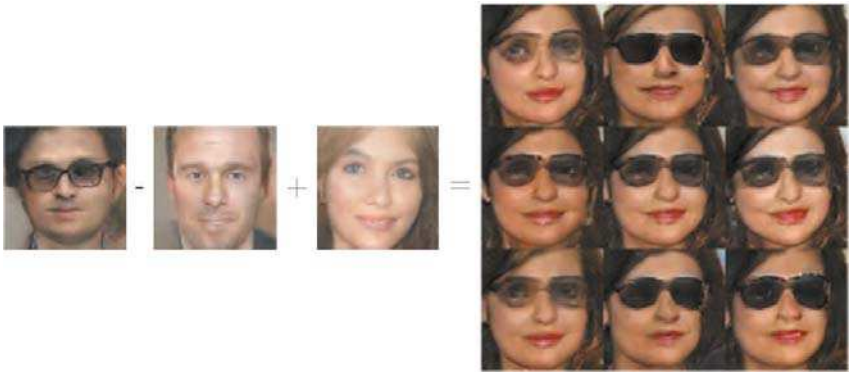

Рис. 15.9 ❖ Порождающая модель обучилась распределенному представлению, которое разделяет концепции пола и ношения очков. Если начать с представления концепции мужчины в очках, затем вычесть вектор, представляющий концепцию мужчины без очков, и, наконец, прибавить вектор, представляющий концепцию женщины без очков, то мы получим вектор, представляющий концепцию женщины в очках. Порождающая модель корректно декодирует все эти представляющие векторы в изображения, которые можно распознать как члены правильного класса. Изображения взяты из работы Radford et al. (2015) с разрешения авторов

15.5. Экспоненциальный выигрыш от глубины

В разделе 6.4.1 мы видели, что многослойные перцептроны являются универсальными аппроксиматорами и что некоторые функции можно представить экспоненциально меньшими глубокими сетями, сравнимыми с мелкими сетями. Такое уменьшение размера модели ведет к улучшению статистической эффективности. В этом разделе мы опишем обобщение подобных результатов на другие виды моделей с распределенными скрытыми представлениями.

В разделе 15.4 был приведен пример порождающей модели, которая обучилась факторам, объясняющим изображения лиц: пол человека и ношение очков. Эта порождающая модель была основана на глубокой нейронной сети. Было бы странно ожидать, что мелкая сеть, например линейная, сможет обучиться сложной связи между абстрактными объясняющими факторами и пикселями изображения. В этой и других задачах ИИ факторы, которые выбираются почти независимо друг от друга, скорее всего, будут очень высокого уровня и связаны с входными данными нелинейно. Мы утверждаем, что для этого необходимы *глубокие* распределенные представления, в которых высокоуровневые признаки (рассматриваемые как функции входа) или факторы (рассматриваемые как порождающие причины) получаются в результате композиции большого числа нелинейностей.

Для многих ситуаций было доказано, что организация вычислений посредством композиции многих нелинейностей и иерархии повторно используемых признаков может дать экспоненциальный прирост статистической эффективности помимо экспоненциального же прироста за счет использования распределенного представления. Можно показать, что многие виды сетей (в т. ч. с насыщающими нелинейностями, булевыми вентилями, суммами-произведениями или радиально-базисными блоками) с одним скрытым слоем являются универсальными аппроксиматорами. Такое семей-

ство моделей может аппроксимировать широкий класс функций (включающий все непрерывные функции) с произвольной точностью. Однако необходимое для этого число скрытых блоков может быть очень велико. Есть теоретические результаты о выразительной мощности глубоких архитектур, согласно которым существуют семейства функций, допускающих эффективное представление архитектурой глубины k, но для этого требуется экспоненциально большое число скрытых блоков (относительно размера входа) с недостаточной глубиной (2 или $k - 1$).

В разделе 6.4.1 мы видели, что детерминированные сети прямого распространения являются универсальными аппроксиматорами функций. Многие структурные вероятностные модели с одним скрытым слоем, в т. ч. ограниченные машины Больцмана и глубокие сети доверия, являются универсальными аппроксиматорами распределений вероятности (Le Roux and Bengio, 2008, 2010; Montúfar and Ay, 2011; Montúfar, 2014; Krause et al., 2013).

В разделе 6.4.1 мы видели, что достаточно глубокая сеть прямого распространения может давать экспоненциальный выигрыш по сравнению со слишком мелкой сетью. Подобные результаты можно получить и для других моделей, например вероятностных. Одна из таких вероятностных моделей – **сеть сумм и произведений** (sum-product network – SPN) (Poon and Domingos, 2011). В этих моделях используются полиномиальные цепочки для вычисления распределения вероятности множества случайных величин. В работе Delalleau and Bengio (2011) показано, что существуют распределения вероятности, для которых требуется некоторая минимальная глубина SPN, чтобы избежать экспоненциально большой модели. В более поздней работе Martens and Medabalimi (2014) показано, что существуют значительные различия между любыми двумя конечными значениями глубины SPN и что некоторые ограничения, которые вводят для того, чтобы с SPN можно было практически работать, могут ограничить их репрезентативную способность.

Интересны также теоретические результаты по выразительной мощности глубоких контуров, связанных со сверточными сетями, которые демонстрируют экспоненциальный выигрыш глубокого контура, даже когда мелкий контур разрешено использовать только для аппроксимации функции, вычисленной глубоким контуром (Cohen et al., 2015). Для сравнения отметим, что в предыдущей теоретической работе рассматривался только случай, когда мелкий контур обязан точно реплицировать конкретные функции.

15.6. Ключ к выявлению истинных причин

В заключение этой главы вернемся к одному из исходных вопросов: благодаря чему одно представление оказывается лучше другого? В разделе 15.3 мы дали один ответ: идеальным является представление, которое разделяет каузальные факторы вариативности данных, особенно те, что напрямую относятся к нашему приложению. Большинство стратегий обучения представлений основано на введении ключей, которые помогают процедуре обучения находить факторы вариативности. Ключи могут помочь обучаемому отделить эти наблюдаемые факторы от остальных. В обучении с учителем имеется очень сильный ключ: метка $\boldsymbol{y}$, сопровождающая каждый вход $\boldsymbol{x}$; обычно она непосредственно задает значение, по меньшей мере, одного фактора вариативности. В более общем случае, чтобы воспользоваться большим объемом непомеченных данных, процедура обучения представления пользуется другими, не столь прямыми подсказками, касающимися каузальных факторов. Эти подсказки могут иметь вид

априорных предположений, вводимых проектировщиками алгоритма, для того чтобы направить обучаемого в нужную сторону. Такие результаты, как теорема об отсутствии бесплатных завтраков, показывают, что для достижения хорошей обобщаемости необходима какая-то стратегия регуляризации. Конечно, найти превосходную во всех отношениях стратегию регуляризации невозможно, но одна из целей глубокого обучения – отыскать набор достаточно общих стратегий, применимых к широкому спектру задач ИИ типа тех, с которыми легко справляются люди и животные.

Ниже приведен список этих общих стратегий регуляризации. Конечно, он не исчерпывающий, но содержит конкретные примеры того, как алгоритм обучения можно направить на выявление признаков, соответствующих каузальным факторам. Этот список приведен в разделе 3.1 работы Bengio et al. (2013d) и частично воспроизводится здесь.

- *Гладкость*. Это предположение о том, что $f(\boldsymbol{x} + \varepsilon \boldsymbol{d}) \approx f(\boldsymbol{x})$ для единичного вектора $\boldsymbol{d}$ и малого ε. Благодаря ему обучаемая модель может обобщаться на точки в пространстве входов, близкие к обучающим примерам. Эта идея эксплуатируется во многих алгоритмах машинного обучения, но ее недостаточно для преодоления проклятия размерности.
- *Линейность*. Во многих алгоритмах машинного обучения предполагается, что связи между некоторыми переменными линейны. Это позволяет алгоритму давать предсказания даже в областях, очень далеких от наблюдаемых данных, но иногда предсказания получаются излишне экстремальными. В большинстве простых алгоритмов, не предполагающих гладкости, делается предположение о линейности. В действительности это разные предположения – линейные функции с большими весами, применяемые в пространствах высокой размерности, могут быть не слишком гладкими. Дальнейшее обсуждение ограничений предположения о линейности см. в работе Goodfellow et al. (2014b).
- *Несколько объясняющих факторов*. В основе многих алгоритмов обучения представлений лежит предположение о множественности факторов, объясняющих данные, и о том, что большинство задач легко решается, если известно состояние каждого фактора. В разделе 15.3 описано, как такой взгляд на вещи обосновывает обучение с частичным привлечением учителя посредством обучения представлений. Чтобы обучиться структуре $p(\boldsymbol{x})$, нужно обучиться некоторым признакам, полезным для моделирования $p(\boldsymbol{y} \mid \boldsymbol{x})$, потому что то и другое зависит от одних и тех же объясняющих факторов. В разделе 15.4 описано, как этот взгляд обосновывает использование распределенных представлений, когда разные направления в пространстве представлений соответствуют разным факторам вариативности.
- *Каузальные факторы*. Модель строится так, чтобы факторы вариативности, описываемые обученным представлением $\boldsymbol{h}$, рассматривались как причины наблюдаемых данных $\boldsymbol{x}$, а не наоборот. Как было сказано в разделе 15.3, это полезно для обучения с частичным привлечением учителя и делает обученную модель более устойчивой, когда распределение причин изменяется или когда модель применяется к решению новой задачи.
- *Глубина, или иерархическая организация объясняющих факторов*. Высокоуровневые абстрактные концепции можно определить в терминах более простых концепций, образующих иерархию. С иной точки зрения, использование глубокой архитектуры выражает нашу веру в то, что задачу следует решать с помощью многошаговой программы, каждый шаг которой опирается на результаты обработки, произведенной на предыдущих шагах.

- *Факторы, разделяемые между задачами.* Когда имеется много задач, соответствующих разным переменным y_i, разделяющим общий вход $\mathbf{x}$, или когда каждая задача ассоциирована с подмножеством или функцией $f^{(i)}(\mathbf{x})$ глобального входа $\mathbf{x}$, предполагается, что каждая y_i ассоциирована со своим подмножеством общего пула релевантных факторов $\mathbf{h}$. Поскольку эти подмножества пересекаются, обучение всех $P(y_i \mid \mathbf{x})$ посредством промежуточного разделяемого представления $P(\mathbf{h} \mid \mathbf{x})$ дает возможность разделить статистическую силу между задачами.
- *Многообразия.* Масса вероятности имеет тенденцию концентрироваться, а области, в которых она концентрируется, локально связны и занимают очень малый объем. В непрерывном случае эти области можно аппроксимировать многообразиями гораздо меньшей размерности, чем исходное пространство данных. Многие алгоритмы машинного обучения разумно ведут себя только на таком многообразии (Goodfellow et al., 2014b). Некоторые алгоритмы, особенно автокодировщики, пытаются явно обучиться структуре такого многообразия.
- *Естественная кластеризация.* Во многих алгоритмах машинного обучения предполагается, что каждому связному многообразию в пространстве входов можно сопоставить один класс. Данные могут располагаться на нескольких несвязных многообразиях, но внутри каждого из них класс остается постоянным. Это предположение лежит в основе разнообразных алгоритмов обучения, включая касательное распространение, двойное обратное распространение, классификатор по касательной к многообразию и состязательное обучение.
- *Временная и пространственная когерентность.* Анализ медленных признаков и родственные алгоритмы предполагают, что наиболее важные объясняющие факторы изменяются во времени медленно или, по крайней мере, что истинные объясняющие факторы предсказать легче, чем сами наблюдения, например значения пикселей. Дополнительные детали см. в разделе 13.3.
- *Разреженность.* Предполагается, что большинство признаков не должно быть релевантно описанию большинства входов, т. е. нет нужды использовать признак, обнаруживающий большегрузный грузовик, если мы представляем изображение кошки. Поэтому разумно наложить априорное ограничение: любой признак, который можно интерпретировать как «присутствует» или «отсутствует», в большинстве случаев принимает значение «отсутствует».
- *Простота зависимостей между факторами.* В хорошем высокоуровневом представлении факторы связаны между собой простыми зависимостями. Простейшая из них – безусловная независимость, $P(\mathbf{h}) = \prod_i P(\mathbf{h}_i)$, но линейные зависимости или зависимости, улавливаемые мелким автокодировщиком, также разумны. Это предположение, имеющее место во многих физических законах, делается, когда поверх обученного представления реализуется линейный предиктор или факторизованное априорное распределение.

Идея обучения представлений связывает воедино многие формы глубокого обучения. Сети прямого распространения и рекуррентные сети, автокодировщики и глубокие вероятностные модели – все они обучают представления и используют их. Обучение наилучшего из возможных представлений остается увлекательным направлением исследований.

Глава 16

Структурные вероятностные модели в глубоком обучении

Глубокое обучение опирается на многочисленные формализмы моделирования, которыми исследователи могут пользоваться при проектировании и описании алгоритмов. Один из таких формализмов – идея **структурной вероятностной модели**. Мы уже кратко обсуждали их в разделе 3.14, и этого обсуждения было достаточно, чтобы понять, как структурные вероятностные модели используются в качестве языка для описания некоторых алгоритмов. Теперь же, в третьей части книги, такие модели предстанут в виде ключевого ингредиента многих из наиболее важных направлений исследований в глубоком обучении. Эта глава задумана независимой от остальной книги: для ее чтения не придется обращаться к ранее изложенному материалу.

Структурная вероятностная модель – это способ описания распределения вероятности с помощью графа, показывающего, какие случайные величины взаимодействуют между собой непосредственно. Здесь слово «граф» употребляется в смысле теории графов и обозначает множество вершин, соединенных ребрами. Поскольку структура модели определена в виде графа, такие модели часто называют также **графическими**.

Ученые, работающие в этой области, разработали много разных моделей, алгоритмов обучения и алгоритмов вывода. В этой главе мы познакомимся с основными идеями, делая упор на концепциях, которые оказались особенно полезны в контексте глубокого обучения. Если вы уже хорошо знакомы с графическими моделями, то можете спокойно пропустить большую часть главы. Однако даже специалисту будет небезынтересно прочитать последний раздел 16.7, в котором описываются некоторые специфические способы применения графических моделей в алгоритмах глубокого обучения. В глубоком обучении структуры моделей алгоритмы обучения и процедуры вывода используются совсем не так, как принято среди исследователей графических моделей. В этой главе мы расскажем об этих отличиях и объясним, в чем причина.

Сначала поговорим о проблемах, возникающих при построении крупномасштабных вероятностных моделей, а затем покажем, как использовать граф для описания структуры распределения вероятности. Хотя этот подход позволяет преодолеть многие трудности, у него есть собственные проблемы. Одна из главных трудностей графического моделирования – понять, какие величины должны взаимодействовать

непосредственно, т. е. какие графовые структуры лучше всего подходят для данной задачи. В разделе 16.5 мы наметим два подхода к решению этой проблемы, основанных на понятии зависимости. И в разделе 16.7 завершим обсуждение, рассказав, каким подходам к графическому моделированию отдают предпочтение специалисты по глубокому обучению.

16.1. Проблема бесструктурного моделирования

Цель глубокого обучения состоит в том, чтобы масштабировать методы машинного обучения на задачи, возникающие в области искусственного интеллекта. Это означает, что мы должны понимать данные высокой размерности с развитой структурой. Например, мы хотели бы, чтобы алгоритмы ИИ понимали естественные изображения[1], звуковые сигналы, соответствующие речи, и документы, содержащие много слов и знаков препинания.

Алгоритмы классификации могут получать входные данные из такого многомерного распределения и сопоставлять им обобщающую метку категории: какой объект изображен на фотографии, какое слово произнесено, какой теме посвящен документ. В процессе классификации большая часть присутствующей во входных данных информации отбрасывается и порождается единственный выход (или распределение вероятности значений выхода). Кроме того, классификатор часто способен игнорировать многие части входа. Например, при распознавании объекта на фотографии мы обычно можем игнорировать фон.

Вероятностную модель можно попросить выполнить эти и многие другие задачи. Зачастую они обходятся дороже классификации. Некоторые из них порождают несколько выходных значений. Большинство требует полного понимания структуры входных данных в целом, не допуская игнорирования отдельных участков. К числу таких задач относятся следующие:

- **оценка плотности.** По входу $\boldsymbol{x}$ система машинного обучения возвращает оценку истинной плотности $p(\boldsymbol{x})$ порождающего данные распределения. Результатом является единственное значение, но требуется полное понимание входных данных. Если хотя бы один элемент вектора необычен, система должна назначить всему вектору низкую вероятность;
- **шумоподавление.** Видя поврежденный или искаженный в результате наблюдения входной сигнал $\tilde{\boldsymbol{x}}$, система машинного обучения возвращает оценку исходного или правильного сигнала $\boldsymbol{x}$. Например, систему машинного обучения можно попросить удалить пыль или царапины со старой фотографии. Результатом будет много выходов (каждый элемент очищенного от шумов примера $\boldsymbol{x}$), и требуется понимание всего входа (потому что даже при наличии всего одного поврежденного участка окончательная оценка будет «поврежден»);
- **восполнение отсутствующих значений.** Видя результаты наблюдений некоторых элементов $\boldsymbol{x}$, модель возвращает оценку распределения вероятности некоторых или всех ненаблюдавшихся элементов $\boldsymbol{x}$. Выходов получается несколько. Поскольку может потребоваться восстановить произвольные элементы $\boldsymbol{x}$, модель должна понимать структуру всего входа;

---

[1] **Естественным** называется изображение, снятое камерой в обычных окружающих условиях, в отличие от синтезированного изображения, снимка веб-страницы и т. п.

- **выборка.** Модель генерирует новую выборку из распределения $p(\boldsymbol{x})$. К таким приложениям относится синтез речи, т. е. порождение звуковых сигналов, похожих на естественную человеческую речь. Необходимы несколько выходных значений и хорошая модель всего входа. Если в выборке присутствует хотя бы один элемент из неправильного распределения, то результат всей выборки не годится.

Пример задачи выборки с использованием небольших естественных изображений приведен на рис. 16.1.

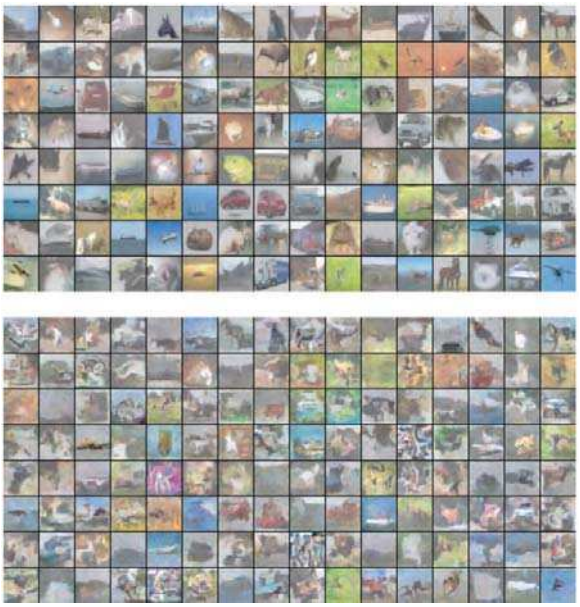

Рис. 16.1 ❖ Вероятностное моделирование естественных изображений. (*Сверху*) Примеры цветных изображений 32×32 из набора данных CIFAR-10 (Krizhevsky and Hinton, 2009). (*Снизу*) Выборка из структурной вероятностной модели, обученной на этом наборе данных. Каждый элемент выборки находится в той же позиции сетки, что и обучающий пример, ближайший к нему в евклидовом пространстве. Это сопоставление показывает, что модель действительно синтезирует новые изображения, а не просто запоминает обучающие данные. Контрастность обоих наборов изображений подобрана для дисплея. Рисунок взят из работы Courville et al. (2011)

Моделирование нетривиального распределения тысяч или миллионов случайных величин – трудная задача как с вычислительной, так и со статистической точки зрения. Предположим, что требуется моделировать только бинарные величины. Даже этот простейший случай вызывает, на первый взгляд, непреодолимые сложности. Для небольшого цветного (RGB) изображения размера 32×32 существуют 2^{3072} возможных бинарных изображений. Это число в 10^{800} больше числа атомов во Вселенной.

В общем случае для моделирования случайного вектора $\mathbf{x}$, содержащего n дискретных величин, каждая из которых принимает k значений, при наивном подходе к представлению $P(\mathbf{x})$ потребуется хранить таблицу вероятностей каждого возможного выхода, т. е. k^n параметров!

Это невозможно по нескольким причинам:

- *память – стоимость хранения представления*. Для сколько-нибудь больших n и k в таблице, представляющей распределение, придется хранить слишком много значений;
- *статистическая эффективность*. С увеличением числа параметров модели возрастает и объем обучающих данных, необходимых для выбора значений этих параметров с помощью статистического оценивателя. Поскольку в табличной модели число параметров астрономически велико, для ее точной аппроксимации понадобится столь же гигантский обучающий набор. Любая такая модель окажется сильно переобученной, если не ввести дополнительные предположения о связях между элементами таблицы (как, например, в возвратных или сглаженных n-граммных моделях, см. раздел 12.4.1);
- *этап выполнения – стоимость вывода*. Пусть мы хотим произвести вывод, используя модель совместного распределения $P(\mathbf{x})$ для вычисления какого-то другого распределения, например маргинального распределения $P(\mathrm{x}_1)$ или условного распределения $P(\mathrm{x}_2 \mid \mathrm{x}_1)$. Для вычисления этих распределений понадобится выполнить суммирование по всей таблице, поэтому стоимость таких операций так же недопустимо высока, как и стоимость хранения модели;
- *этап выполнения – стоимость выборки*. Предположим, что требуется произвести выборку из модели. Наивный способ – выбрать какое-то значение $\mathrm{u} \sim U(0, 1)$, а затем обходить таблицу, складывая значения вероятностей до тех пор, пока сумма не превзойдет u, после чего вернуть в качестве результата элемент в соответствующей позиции таблицы. Но в худшем случае для этого понадобится прочитать всю таблицу, так что стоимость этой операции, как и прочих, экспоненциально высока.

Проблема табличного подхода в том, что мы явно моделируем все возможные взаимодействия между всеми возможными подмножествами величин. Распределения вероятности, встречающиеся в реальных задачах, гораздо проще. Обычно большинство величин влияет друг на друга только косвенно.

Рассмотрим, к примеру, моделирование времени финиширования команды в эстафете. Пусть команда состоит из трех человек: Анна, Борис и Вера. В начале эстафеты палочка находится у Анны, которая начинает бежать по дорожке. Завершив свой этап, она передает палочку Борису. Борис бежит свой этап и передает палочку Вере, которой выпал последний этап. Мы можем смоделировать время финиширования каждого участника команды с помощью непрерывной случайной величины. Время финиширования Анны не зависит от других участников, поскольку она бежит первой. Время финиширования Бориса зависит от Анны, потому что Борис не может начать свой этап, пока Анна не придет к финишу. Если Анна прибежит быстрее, то при прочих равных условиях и Борис финиширует быстрее. Наконец, время финиширования Веры зависит от обоих ее товарищей по команде. Если Анна бежит медленно, то и Борис, вероятно, финиширует слишком поздно. Следовательно, Вера поздно начнет бежать и, скорее всего, поздно придет к финишу. Однако время финиширования Веры зависит от времени финиширования Анны лишь косвенно – через Бориса. Если мы уже знаем время финиширования Бориса, то не получим более точной оценки времени финиширования Веры, узнав, когда финишировала Анна. Таким образом, для моделирования эстафеты достаточно всего двух взаимодействий: влияние Анны на Бориса и влияние Бориса на Веру. Третье, косвенное взаимодействие между Анной и Верой из модели можно исключить.

Структурные вероятностные модели предлагают формальную систему моделирования только прямых взаимодействий между случайными величинами. Это позволяет обойтись значительно меньшим числом параметров модели и, следовательно, получить надежную оценку при меньшем объеме данных. Уменьшение размера модели также кардинально снижает вычислительную стоимость с точки зрения памяти для хранения модели, времени выполнения вывода и выборки из модели.

16.2. Применение графов для описания структуры модели

В структурных вероятностных моделях для представления взаимодействий между случайными величинами применяются графы. Вершины представляют случайные величины, а ребра – прямые взаимодействия. Прямые взаимодействия подразумевают также наличие косвенных, но явно моделируются только первые.

Существует несколько способов описать взаимодействия случайных величин с помощью графа. В следующих разделах мы опишем несколько наиболее популярных и полезных подходов. Графические модели можно разбить на две большие категории: основанные на ориентированных ациклических графах и основанные на неориентированных графах.

16.2.1. Ориентированные модели

Один из видов структурных вероятностных моделей – **ориентированная графическая модель**, известная также под названиями **сеть доверия** или **байесовская сеть**[1] (Pearl, 1985).

Ориентированные графические модели называются «ориентированными», потому что ребра графа ориентированы, т. е. направлены от одной вершины к другой. Направление ребра обозначается стрелкой. Стрелка показывает, что распределение вероятности одной величины определяется в терминах другой. Наличие стрелки, направленной от a к b, означает, что распределение вероятности b определено с помощью условного распределения, причем a – одна из величин, находящихся справа от вертикальной черты. Иными словами, распределение b зависит от значения a.

Продолжая пример эстафеты из раздела 16.1, обозначим время финиширования Анны t_0, Бориса – t_1, а Веры – t_2. Как мы уже видели, оценка t_1 зависит от t_0. Оценка t_2 прямо зависит от t_1, но лишь косвенно от t_0. Мы можем изобразить эти связи в виде ориентированной графической модели, показанной на рис. 16.2.

Формально ориентированная графическая модель над величинами $\mathbf{x}$ определяется ориентированным ациклическим графом $\mathcal{G}$, вершины которого соответствуют случайным величинам модели, и множеством **локальных условных распределений вероятности** $p(x_i \mid Pa_{\mathcal{G}}(x_i))$, где $Pa_{\mathcal{G}}(x_i)$ обозначает множество родителей x_i в $\mathcal{G}$. Распределение вероятности $\mathbf{x}$ имеет вид

$$p(\mathbf{x}) = \prod_i p(x_i \mid Pa_{\mathcal{G}}(x_i)). \tag{16.1}$$

---

[1] Джуда Пирл предлагал использовать термин «байесовская сеть», когда нужно «подчеркнуть оценочную» природу значений, вычисляемых сетью, т. е. тот факт, что они представляют степень веры, а не частоты событий.

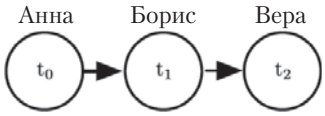

Рис. 16.2 ❖ Ориентированная графическая модель эстафеты. Время финиширования Анны t_0 влияет на время финиширования Бориса t_1, поскольку Борис не начинает бежать, пока Анна не придет к финишу. Аналогично Вера начинает бежать только после того, как Борис финиширует, поэтому время финиширования Бориса t_1 прямо влияет на время финиширования Веры t_2

В нашем примере эстафеты, описываемом графом на рис. 16.2, это означает, что:

$$p(t_0, t_1, t_2) = p(t_0)p(t_1 \mid t_0)p(t_2 \mid t_1). \tag{16.2}$$

Это наше первое знакомство со структурной вероятностной моделью в действии. Изучив стоимость ее использования, мы увидим, что структурное моделирование имеет много преимуществ по сравнению с бесструктурным.

Предположим, что мы дискретизировали интервал времени от нулевой до десятой минуты, разбив его на 6-секундные промежутки. Тогда каждая из случайных величин t_0, t_1 и t_2 может принимать 100 различных значений. Если бы мы попытались представить $p(t_0, t_{01}, t_2)$ таблицей, то понадобилось бы хранить 999 999 значений (100 значений t_0 × 100 значений t_1 × 100 значений t_2 минус 1, поскольку вероятность одной конфигурации можно не задавать, т. к. сумма всех вероятностей должна быть равна 1). Если же завести по одной таблице для каждого условного распределения, то для хранения распределения t_0 нужно 99 значений, для хранения t_1 при условии t_0 – 9900 значений и столько же для хранения t_2 при условии t_1. Всего получается 19 899 значений. То есть применение ориентированной графической модели позволило уменьшить число параметров больше чем в 50 раз!

В общем случае для моделирования n дискретных величин, каждая из которых принимает k значений, решение с одной таблицей имеет порядок $O(k^n)$, как мы видели раньше. Теперь предположим, что мы построили ориентированную графическую модель над этими величинами. Если m – максимальное число величин по любую сторону от вертикальной черты в одном условном распределении вероятности, то стоимость хранения таблиц для ориентированной модели имеет порядок $O(k^m)$. Если удастся спроектировать модель, в которой $m << n$, то мы получим очень существенное улучшение.

Иными словами, если у каждой величины в графе мало родителей, то для представления распределения нужно очень мало параметров. Введя дополнительные ограничения на структуру графа, например потребовав, что он был деревом, мы также сможем гарантировать, что такие операции, как вычисление маргинального или условного распределения подмножеств величин, производятся эффективно.

Важно понимать, какие виды информации можно закодировать в виде графа, а какие нельзя. В графе представлены только упрощающие предположения о том, какие величины условно независимы. Можно также вводить другие типы упрощающих предположений. Предположим, к примеру, что Борис всегда бежит одинаково вне зависимости от результата, показанного Анной. (На практике результат Анны, скорее всего, влияет на результат Бориса – увидев, что Анна пробежала необычно быстро, Борис может воодушевиться и сделать все возможное, чтобы не уступить ей

или, наоборот, решить, что дело сделано, и бежать спустя рукава). Тогда единственное влияние Анны на время финиширования Бориса заключается в том, что нужно прибавить время Анны к заранее известному времени Бориса. Это наблюдение позволяет определить модель с $O(k)$ параметрами вместо $O(k^2)$. Отметим, однако, что t_0 и t_1 попрежнему прямо зависят друг от друга, т. к. t_1 представляет абсолютное время финиширования Бориса, а не время, потраченное им на свой этап. Это значит, что в графе должна остаться стрелка из t_0 в t_1. Предположение, что личное время Бориса ни от чего не зависит, невозможно закодировать в графе с вершинами t_0, t_1 и t_2. Вместо этого мы должны закодировать эту информацию в определении самого условного распределения. Теперь условное распределение не является таблицей с $k \times k - 1$ элементами, индексированной величинами t_0 и t_1, а описывается несколько более сложной формулой всего с $k - 1$ параметрами. Синтаксис ориентированной графической модели не налагает никаких ограничений на способ определения условных распределений. Он лишь определяет, какие случайные величины могут входить в них в качестве аргументов.

16.2.2. Неориентированные модели

Ориентированные графические модели предлагают один из возможных языков описания структурных вероятностных моделей. Другой популярный язык – **неориентированные модели**, или **марковские случайные поля**, или **марковские сети** (Kindermann, 1980). Как следует из самого названия, в неориентированных моделях используются графы с неориентированными ребрами.

Ориентированные модели естественны в ситуациях, когда есть очевидные причины нарисовать стрелку в определенном направлении. Так часто бывает, когда мы понимаем причинно-следственные связи, и эти связи однонаправленные. Пример такой ситуации – моделирование эстафеты. Участники, бегущие раньше, оказывают влияние на время финиширования бегущих позже, а те, кто бежит позже, никак не влияют на результат уже пробежавших.

Но не во всех интересующих нас ситуациях можно однозначно определить направление взаимодействия. Когда у взаимодействия нет очевидного направления или когда оно действует в обоих направлениях, лучше использовать неориентированную модель. В качестве примера рассмотрим модель распределения с тремя бинарными величинами: вы больны или нет, ваш коллега болен или нет и ваш сосед по комнате болен или нет. Как и в примере с эстафетой, мы можем принять упрощающие предположения об имеющихся взаимодействиях. В предположении, что коллега и сосед незнакомы, крайне маловероятно, что один мог заразить другого напрямую. Это событие можно счесть настолько редким, что моделировать его бессмысленно. Однако вполне вероятно, что каждый из них мог заразить вас и что вы могли передать заразу другому. Мы можем смоделировать косвенную передачу болезни от коллеги соседу, смоделировав передачу от коллеги вам и от вас соседу.

В этом случае с равным успехом как вы можете быть причиной заболевания соседа, так и он – вашего, поэтому очевидного направления взаимодействия не существует. Следовательно, напрашивается применение неориентированной модели. Как и в ориентированных моделях, если две вершины соединены ребром, то соответствующие им случайные величины взаимодействуют непосредственно. Но, в отличие от ориентированной модели, в неориентированной на ребре нет стрелки, и с ним не ассоциировано никакое условное распределение вероятности.

Обозначим случайную величину, представляющую состояние вашего здоровья, h_y, состояние здоровья вашего соседа h_r, а состояние здоровья коллеги – h_c. Граф, описывающий эту ситуацию, показан на рис. 16.3.

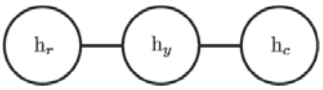

Рис. 16.3 ❖ Ориентированный граф, представляющий взаимное влияние состояния здоровья вашего коллеги h_r, ваше h_y и вашего соседа h_c. Вы и ваш сосед могли заразить друг друга, и то же самое относится к коллеге. Но в предположении, что коллега и сосед незнакомы, они могли заразить друг друга только через вас

Формально говоря, неориентированная графическая модель – это структурная вероятностная модель, определенная над неориентированным графом $\mathcal{G}$. Для каждой клики $\mathcal{C}$ графа[1] **фактор** $\phi(\mathcal{C})$ (или **потенциал клики**) измеряет склонность случайных величин, входящих в эту клику, к пребыванию в каждом из возможных совместных состояний. Факторы по определению неотрицательны. Вместе они образуют **ненормированное распределение вероятности**

$$\tilde{p}(\mathbf{x}) = \prod_{\mathcal{C}\in\mathcal{G}} \phi(\mathcal{C}). \tag{16.3}$$

Работа с этим ненормированным распределением будет эффективна при условии, что клики малы. Его идея в том, что состояния с высокой взаимной склонностью более вероятны. Однако, в отличие от байесовской сети, в определении клик нет никакой особенной структуры, поэтому нет гарантии, что их перемножение даст корректное распределение вероятности. На рис. 16.4 приведен пример чтения информации о факторизации из неориентированного графа.

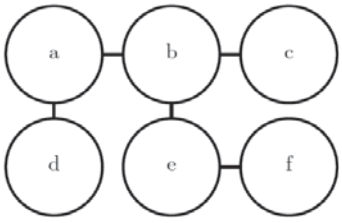

Рис. 16.4 ❖ Из этого графа следует, что распределение $p(a, b, c, d, e, f)$ можно записать в виде $(1/Z)\phi_{a,b}(a, b)\phi_{b,c}(b, c)\phi_{a,d}(a, d)\phi_{b,e}(b, e)\phi_{e,f}(e, f)$ при подходящем выборе функций ϕ

В примере с распространением болезни между тремя лицами есть две клики. Одна из них содержит h_y и h_c. Фактор этой клики можно определить с помощью таблицы такого вида:

	$h_y = 0$	$h_y = 1$
$h_c = 0$	2	1
$h_c = 1$	1	10

[1] Кликой называется подмножество вершин графа, любые две из которых соединены ребром.

Состояние 1 означает, что человек здоров, а состояние 0 – что он болен (подхватил простуду). И вы, и ваш коллега обычно здоровы, поэтому склонность соответствующего состояния максимальна. Состояние, в котором только один из вас болен, имеет наименьшую склонность, поскольку оно встречается редко. У состояния, в котором вы оба больны (поскольку один заразил другого), склонность выше, хотя оно встречается не столь часто, как состояние, когда вы оба здоровы.

Чтобы завершить построение модели, нужно еще определить аналогичный фактор для клики, содержащей h_y и h_r.

16.2.3. Статистическая сумма

Для ненормированного распределения вероятности гарантируется, что оно всюду неотрицательно, но не гарантируется, что сумма или интеграл равны 1. Чтобы получить корректное распределение вероятности, необходимо произвести нормировку[1]:

$$p(\mathbf{x}) = \frac{1}{Z}\tilde{p}(\mathbf{x}), \tag{16.4}$$

где коэффициент Z обеспечивает равенство суммы вероятностей или интеграла плотности единице:

$$Z = \int \tilde{p}(\mathbf{x})d\mathbf{x}. \tag{16.5}$$

Z можно считать константой, если функции ϕ постоянны. Отметим, что если у функций ϕ есть параметры, то Z – функция этих параметров. В литературе принято записывать Z без параметров для экономии места. Нормировочная постоянная Z называется **статистической суммой**, этот термин заимствован из статистической физики.

Поскольку Z – это интеграл или сумма по всем возможным совместным распределениям состояния $\mathbf{x}$, вычислить его зачастую невозможно. Чтобы все же получить нормированное распределение вероятности неориентированной модели, структура модели и определения функций ϕ должны допускать эффективное вычисление Z. В контексте глубокого обучения точное вычисление Z – обычно неразрешимая задача. А раз так, то мы должны прибегнуть к аппроксимациям. Такие приближенные алгоритмы – тема главы 18.

При проектировании неориентированных моделей важно иметь в виду, что задать факторы можно таким образом, что Z вообще не существует. Это происходит, если некоторые случайные величины в модели непрерывны, а интеграл $\tilde{p}$ по их области определения расходится. Предположим, к примеру, что мы хотим построить модель одной скалярной величины $x \in \mathbb{R}$ с одним потенциалом клики $\phi(x) = x^2$. В таком случае

$$Z = \int x^2 dx. \tag{16.6}$$

Поскольку этот интеграл расходится, не существует распределения вероятности, соответствующего такому выбору $\phi(x)$. Иногда от выбора некоторого параметра функций ϕ зависит, определено ли распределение вероятности. Например, для $\phi(x; \beta) = \exp(-\beta x^2)$ параметр β определяет существование Z. При положитель-

---

[1] Распределение, полученное нормировкой произведений потенциалов клик, называют также **распределением Гиббса**.

ных β получается нормальное распределение x, а при всех остальных ϕ невозможно нормировать.

Одно из основных различий между ориентированным и неориентированным моделированием состоит в том, что ориентированные модели с самого начала определены непосредственно в терминах распределений вероятности, тогда как неориентированные – более свободно, в терминах функций ϕ, которые впоследствии преобразуются в распределения вероятности. Поэтому при работе с этими моделями интуитивные соображения различны. Имея дело с неориентированными моделями, важно помнить, что область определения каждой величины сильно влияет на вид распределения вероятности, соответствующего данному множеству функций ϕ. Рассмотрим, например, n-мерную векторную случайную величину $\mathbf{x}$ и неориентированную модель, параметризованную вектором смещений $\boldsymbol{b}$. Предположим, что для каждого элемента $\mathbf{x}$ имеется одна клика $\phi^{(i)}(x_i) = \exp(b_i x_i)$. Какое распределение вероятности при этом получается? Для ответа на этот вопрос недостаточно информации, потому что мы еще не задали область определения $\mathbf{x}$. Если $\mathbf{x} \in \mathbb{R}^n$, то интеграл, определяющий Z, расходится, и никакого распределения вероятности не существует. Если $\mathbf{x} \in \{0, 1\}^n$, то $p(\mathbf{x})$ разлагается в произведение n независимых распределений: $p(\mathrm{x}_i = 1) = \mathrm{sigmoid}(b_i)$. Если область определения $\mathbf{x}$ – множество элементарных базисных векторов ({[1, 0, ..., 0], [0, 1, ..., 0], ..., [0, 0, ..., 1]}), то $p(\mathbf{x}) = \mathrm{softmax}(\boldsymbol{b})$, поэтому большое значение b_i фактически сокращает $p(\mathrm{x}_j = 1)$ для $j \neq i$. Часто за счет специального подбора области определения случайной величины можно получить сложное поведение из сравнительно простого множества функций ϕ. Практическое применение этой идеи мы рассмотрим в разделе 20.6.

16.2.4. Энергетические модели

Многие интересные теоретические результаты о неориентированных моделях основаны на предположении о том, что $\forall \mathbf{x}\ \tilde{p}(\mathbf{x}) > 0$. Удобный способ наложить такое ограничение – воспользоваться **энергетической моделью** (energy-based model – EBM), в которой

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})), \tag{16.7}$$

а $E(\mathbf{x})$ называется **функцией энергии**. Поскольку $\exp(z)$ положительно для всех z, гарантируется, что при любой функции энергии все состояния $\mathbf{x}$ будут иметь ненулевую вероятность. Возможность выбирать абсолютно произвольную функцию энергии упрощает обучение. Если бы мы обучали потенциалы клик непосредственно, то пришлось бы использовать ограниченную оптимизацию, чтобы задать неизвестно как выбранное минимальное значение вероятности. А обучая функцию энергии, мы можем пользоваться неограниченной оптимизацией[1]. В энергетической модели вероятности могут быть сколь угодно близки к нулю, но никогда не обращаются в нуль.

Распределение вида (16.7) называется **распределением Больцмана**. Поэтому многие энергетические модели называются **машинами Больцмана** (Fahlman et al., 1983; Ackley et al., 1985; Hinton et al., 1984; Hinton and Sejnowski, 1986). Не существует общепринятого соглашения о том, когда называть модель энергетической, а когда – машиной Больцмана. Термин «машина Больцмана» был введен для обозначения мо-

[1] Для некоторых моделей все же приходится прибегать к ограниченной оптимизации, чтобы гарантировать существование Z.

дели, в которой все переменные бинарные, но сегодня так называются многие модели с вещественными переменными, например ограниченные машины Больцмана с усреднением и ковариацией. Хотя в первоначальном определении машины Больцмана охватывали модели с латентными переменными и без них, в наши дни этот термин чаще всего относится к моделям с латентными переменными, а машины Больцмана без латентных переменных обычно называют случайными марковскими полями, или лог-линейными моделями.

Клики в неориентированном графе соответствуют факторам ненормированной функции вероятности. Поскольку $\exp(a)\exp(b) = \exp(a + b)$, это означает, что разные клики соответствуют разным членам функции энергии. Иными словами, энергетическая модель – это просто частный случай марковской сети: операция потенцирования сопоставляет каждому члену функции энергии фактор отдельной клики. На рис. 16.5 показано, как получить функцию энергии по неориентированному графу. Энергетическую модель с функцией энергии, содержащей несколько членов, можно рассматривать как **произведение экспертов** (Hinton, 1999). Каждый член функции энергии соответствует фактору распределения вероятности, его можно считать «экспертом», который определяет, удовлетворяется ли некоторое мягкое ограничение. Каждый эксперт может налагать только одно ограничение, относящееся лишь к проекции случайных величин на пространство низкой размерности, но в сочетании с произведением вероятностей сообщество экспертов налагает сложное ограничение высокой размерности.

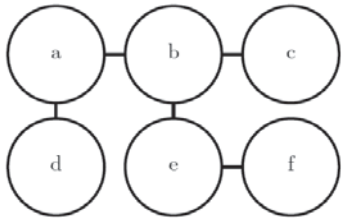

Рис. 16.5 ❖ Из этого графа следует, что $E(a, b, c, d, e, f)$ можно записать в виде $E_{a,b}(a, b) + E_{b,c}(b, c) + E_{a,d}(a, d) + E_{b,e}(b, e) + E_{e,f}(e, f)$ при подходящем выборе функции энергии для каждой клики. Отметим, что функции на рис. 16.4 можно получить, положив каждую равной экспоненте соответствующей отрицательной энергии, например: $\phi_{a,b}(a, b) = \exp(-E(a, b))$

В определении энергетической модели есть часть, не несущая никакой полезной функции с точки зрения машинного обучения: знак – в формуле (16.7). Его можно было бы включить в определение E, поскольку во многих случаях алгоритм обучения вправе выбрать любой знак энергии. Знак – присутствует главным образом для того, чтобы сохранить совместимость между литературой по машинному обучению и по физике. Многими своими достижениями вероятностное моделирование обязано статистической физике, где E обозначает физическую энергию и не может иметь произвольного знака. Термины «энергия» и «статистическая сумма» заимствованы именно оттуда, хотя их математическое применение шире, чем в том физическом контексте, в котором они возникли. Некоторые специалисты по машинному обучению (например, Smolensky [1986], который называл отрицательную энергию **гармонией**) убирали знак минус, но это не стало стандартным соглашением.

Во многих алгоритмах для работы с вероятностными моделями нужно вычислять не $p_{\text{model}}(\boldsymbol{x})$, а только $\log \tilde{p}_{\text{model}}(\boldsymbol{x})$. В энергетических моделях с латентными переменными $\boldsymbol{h}$ такие алгоритмы иногда формулируются в терминах этой величины со знаком минус, называемой **свободной энергией**:

$$\mathcal{F}(\boldsymbol{x}) = -\log \sum_{\boldsymbol{h}} \exp(-E(\boldsymbol{x}, \boldsymbol{h})). \tag{16.8}$$

В этой книге мы предпочитаем более общую формулировку: $\log \tilde{p}_{\text{model}}(\boldsymbol{x})$.

16.2.5. Разделенность и d-разделенность

Ребра в графической модели несут информацию о том, какие переменные взаимодействуют непосредственно. Часто бывает необходимо знать о переменных, взаимодействующих *косвенно*. Некоторые косвенные взаимодействия можно запретить или разрешить посредством наблюдения других переменных. Формально говоря, мы хотели бы знать, какие подмножества переменных условно независимы друг от друга при известных значениях других подмножеств переменных.

Выявить условную независимость в графе неориентированной модели просто. В этом случае условная независимость, вытекающая из структуры графа, называется **разделенностью**. Мы говорим, что множества переменных $\mathbb{A}$ и $\mathbb{B}$ **разделены** третьим заданным множеством переменных $\mathbb{S}$, если из структуры графа следует, что $\mathbb{A}$ не зависит от $\mathbb{B}$ при условии $\mathbb{S}$. Если две переменные a и b соединены путем, содержащим только ненаблюдаемые переменные, то они не являются разделенными. Если между ними вообще нет пути или все соединяющие пути содержат хотя бы одну наблюдаемую переменную, то a и b разделены. Пути, содержащие только ненаблюдаемые переменные, называются «активными», а включающие наблюдаемую переменную – «неактивными».

Наблюдаемые переменные изображаются в графе серым цветом. На рис. 16.6 показано, как при этом выглядят активные и неактивные пути в неориентированной модели, а на рис. 16.7 – как определять разделенность по неориентированному графу.

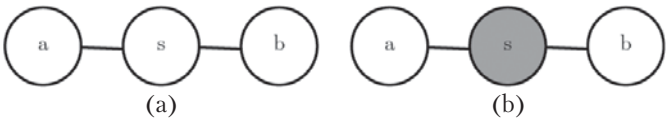

Рис. 16.6 ❖ (*a*) Путь между случайными величинами a и b, проходящий через s, активен, поскольку величина s не наблюдаемая. Это означает, что a и b не разделены. (*b*) Здесь s закрашена серым, т. е. является наблюдаемой. Поскольку единственный путь между a и b проходит через s и этот путь неактивен, можно заключить, что a и b разделены величиной s

Похожие идеи применимы и к ориентированным моделям, только в этом контексте говорят о d-разделенности. Буква «d» означает «dependence» (зависимость). Для ориентированных графов d-разделенность определяется так же, как разделенность для неориентированных: множества переменных $\mathbb{A}$ и $\mathbb{B}$ называются d-разделенными третьим множеством переменных $\mathbb{S}$, если из структуры графа следует, что $\mathbb{A}$ не зависит от $\mathbb{B}$ при условии $\mathbb{S}$.

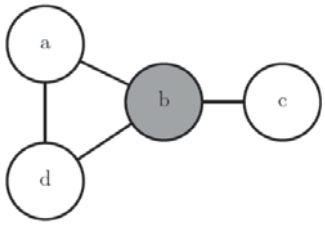

Рис. 16.7 ❖ Пример определения разделенности по неориентированному графу. Здесь b закрашена серым, потому что является наблюдаемой. Поскольку наблюдение b блокирует единственный путь из a в c, можно утверждать, что a и c разделены величиной b. Наблюдение b блокирует также один путь между a и d, но между ними существует другой, активный путь. Следовательно, a и d не разделены b

Как и в случае неориентированных моделей, мы можем исследовать независимость, вытекающую из структуры графа, глядя на активные пути в нем. Как и раньше, две переменные независимы, если между ними существует активный путь, и d-разделенные, если такого пути нет. В ориентированных сетях определить, является ли путь активным, несколько сложнее. Рисунок 16.8 может служить руководством по нахождению активных путей в ориентированной модели. А на рис. 16.9 приведен пример определения некоторых свойств по графу.

Важно помнить, что разделенность и d-разделенность говорят только о тех условных независимостях, *которые следуют из графа*. Не требуется, чтобы из графа вытекали все существующие отношения независимости. В частности, всегда можно использовать полный граф (в котором проведены все возможные ребра) для представления любого распределения. На самом деле некоторые распределения содержат независимости, которые невозможно представить с помощью существующей нотации графов. **Контекстными независимостями** называются такие отношения независимости, которые присутствуют в зависимости от значений некоторых величин в сети. Например, рассмотрим модель с тремя бинарными величинами: a, b и c. Предположим, что при $a = 0$ величины b и c независимы, а при $a = 1$ b всегда равно c. Чтобы представить такое поведение при $a = 1$, необходимо провести ребро между b и c. Но тогда граф не сможет показать, что b и c независимы при $a = 0$.

Вообще, граф никогда не позволяет сделать вывод о наличии независимости, если таковой не существует. Однако могут существовать независимости, не представленные в графе.

16.2.6. Преобразование между ориентированными и неориентированными графами

Мы часто называем конкретную модель машинного обучения ориентированной или неориентированной. Например, ограниченная машина Больцмана обычно считается неориентированной моделью, а разреженное кодирование – ориентированной. Такой выбор слов вносит некоторую путаницу, потому что никакая вероятностная модель не является внутренне ориентированной или неориентированной. Просто одни модели проще описывать с помощью ориентированного графа, а другие – с помощью неориентированного.

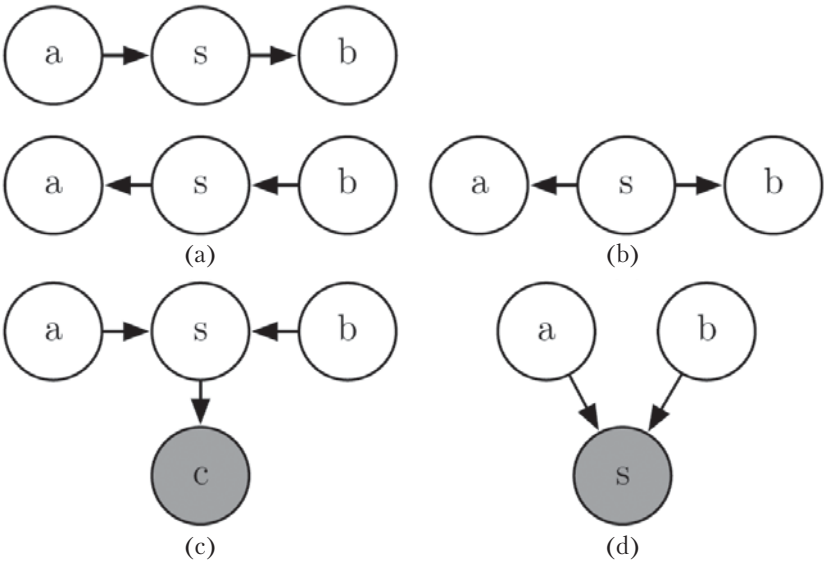

Рис. 16.8 ❖ Все возможные виды активных путей длины 2 между случайными величинами a и b. (*a*) Любой путь со стрелками в одном направлении: от a к b или наоборот. Такой путь будет блокирован, если s – наблюдаемая величина. Мы уже встречались с такими путями в примере эстафеты. (*b*) Величины a и b связаны *общей причиной* s. Допустим, к примеру, что величина s определяет наличие урагана, а величины a и b измеряют скорость ветра в двух соседних метеостанциях. Если на станции a наблюдается очень сильный ветер, то можно ожидать сильного ветра и на станции b. Путь такого вида может блокироваться наблюдением s. Если мы уже знаем, что ураган есть, то можно ожидать сильного ветра в b вне зависимости от того, что наблюдается в a. Если ветер в a слабее ожидаемого (для урагана), то ожидаемая сила ветра в b от этого не изменится (при условии что мы знаем о наличии урагана). Но если s – ненаблюдаемая величина, то a и b зависимы, т. е. путь активен. (*c*) Величины a и b – родители s. Это называется **V-структурой**, или **коллизией**. V-структура связывает a и b путем **оправдания** (explaining away). В данном случае путь активен, когда s наблюдаемая. Предположим, к примеру, что величина s описывает отсутствие вашего коллеги на работе. Величина a говорит, болен ли он, а величина b – находится ли он в отпуске. Если мы наблюдаем, что коллега отсутствует, то можем предположить, что он болен или в отпуске, но маловероятно, что то и другое произошло одновременно. Если выясняется, что он в отпуске, то одного этого факта достаточно для *оправдания* отсутствия. Отсюда мы можем сделать вывод, что коллега, вероятно, не болен. (*d*) Оправдание имеет место, даже когда любой потомок s – наблюдаемая величина! Предположим, к примеру, что величина c представляет факт получения отчета от коллеги. Если вы замечаете, что не получили отчета, то это увеличивает оценку того, что коллеги сегодня нет на работе, из чего, в свою очередь, с вероятностью следует, что он либо болен, либо в отпуске. Путь через V-структуру блокируется только тогда, когда ни один из потомков общей дочерней вершины не является наблюдаемой величиной

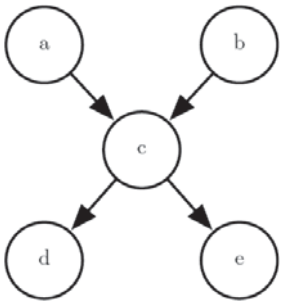

Рис. 16.9 ❖ Из этого графа мы можем получить несколько свойств d-разделенности, например:
- a и b d-разделены пустым множеством;
- a и e d-разделены вершиной c;
- d и e d-разделены вершиной c.

Мы также видим, что некоторые величины перестают быть d-разделенными, если наблюдаются другие величины:
- a и b не являются d-разделенными вершиной c;
- a и b не являются d-разделенными вершиной d

И у ориентированных, и у неориентированных моделей есть свои плюсы и минусы. Ни тот, ни другой подходы нельзя считать наилучшим и предпочтительным во всех случаях. Нужно просто решить, какой язык использовать в каждой задаче. Отчасти выбор определяется тем, какое распределение вероятности мы хотим описать. Предпочтительный вид модели может зависеть от того, какой граф позволяет представить большинство независимостей или в каком графе число ребер меньше. На выбор языка влияют и другие факторы. Даже при работе с одним распределением вероятности иногда выгодно переходить от одного языка моделирования к другому. Иногда другой язык оказывается более подходящим, если мы наблюдаем определенное подмножество переменных или если хотим решить другую вычислительную задачу. Например, ориентированная модель часто позволяет эффективно производить выборку (см. раздел 16.3), а неориентированная бывает полезна для выполнения процедур приближенного вывода (как мы увидим в главе 19, где роль неориентированных моделей выражена формулой (19.56)).

Любое распределение вероятности можно представить ориентированной или неориентированной моделью. В худшем случае распределение всегда можно представить «полным графом». В случае ориентированной модели полным называется любой ориентированный ациклический граф, в котором определено некоторое отношение порядка на случайных величинах, и для каждой величины все предшествующие ей в этом порядке величины являются ее предками в графе. В случае неориентированной модели полным называется просто граф, содержащий единственную клику, в которую входят все величины. Пример показан на рис. 16.10.

Разумеется, полезность графической модели заключается в том, что из структуры графа вытекает, что некоторые величины не взаимодействуют непосредственно. Полный граф не особенно полезен, потому что из него нельзя извлечь никакой информации о независимости.

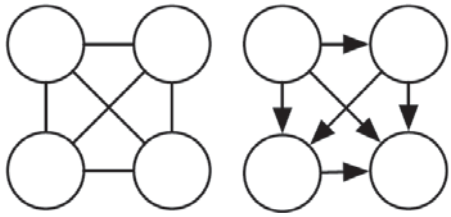

Рис. 16.10 ❖ Примеры полных графов с четырьмя случайными величинами, способных описать любое распределение вероятности. (*Слева*) Полный неориентированный граф; в этом случае существует только один полный граф. (*Справа*) Полный ориентированный граф. В этом случае полных графов много. Мы выбрали некоторое отношение порядка на множестве случайных величин и провели дугу из каждой величины в величину, которая следует за ней в этом порядке. Поэтому число полных ориентированных графов равно факториалу числа случайных величин. В данном случае величины упорядочены слева направо сверху вниз

Представляя распределение вероятности в виде графа, мы хотим выбрать граф так, чтобы независимостей было как можно больше, но только не таких, которых в действительности не существует.

С этой точки зрения, одни распределения более эффективно представляются ориентированными моделями, а другие – неориентированными. Иными словами, в ориентированной модели можно закодировать некоторые независимости, не представимые неориентированной моделью, и наоборот.

В ориентированной модели можно использовать одну структуру специального вида, которую трудно представить в неориентированной модели. Речь идет об **аморальности**. Эта структура имеет место, когда две случайные величины a и b являются родителями третьей величины c, но не существует ребра, соединяющего a и b хотя бы в одном направлении. (Странное название «аморальность» употребляется в литературе по графическим моделям как намек на родителей, не сочетавшихся законным браком.) Для преобразования ориентированной модели с графом $\mathcal{D}$ в неориентированную модель нужно построить новый граф $\mathcal{U}$. Для каждой пары величин x и y в $\mathcal{U}$ включается неориентированное ребро, соединяющее x и y, если в $\mathcal{D}$ существует ориентированное ребро между x и y (в любом направлении) или если x и y – родители третьей величины z в $\mathcal{D}$. Получившийся граф $\mathcal{U}$ называется **моральным графом**. На рис. 16.11 приведены примеры преобразований ориентированных моделей в неориентированные посредством моральзации.

С другой стороны, неориентированные модели могут содержать структуры, не представимые ориентированной моделью. Точнее говоря, ориентированный граф $\mathcal{D}$ не может представить всех условных независимостей, следующих из неориентированного графа $\mathcal{U}$, когда $\mathcal{U}$ содержит **цикл** длины больше 3, если только этот цикл не содержит также **хорду**. Циклом называется последовательность величин, соединенных неориентированными ребрами, такая, что последняя величина соединена с первой. Хордой называется соединение между любыми двумя несоседними величинами в последовательности, определяющей цикл. Если $\mathcal{U}$ содержит циклы длины 4 или больше и не содержит хорд для этих циклов, то перед преобразованием в ориентированную модель мы должны добавить хорды. Добавление хорд приводит к частичному отбрасыванию информации о независимости, закодированной в $\mathcal{U}$. Граф, образованный до-

бавлением хорд в $\mathcal{U}$, называется **хордовым**, или **триангулированным**, поскольку все циклы теперь можно описать в терминах меньших треугольных циклов. Для построения ориентированного графа $\mathcal{D}$ из хордового графа мы должны также назначить каждому ребру направление. Но при этом нужно избегать появления ориентированных циклов в $\mathcal{D}$, потому что иначе получившаяся ориентированная вероятностная модель окажется некорректной. Чтобы назначить ребрам $\mathcal{D}$ направления, мы можем выбрать некоторое отношение порядка на случайных величинах, а затем приписать каждому ребру такое направление, чтобы начальной была меньшая относительно этого порядка величина, а конечной – большая. Это иллюстрируется на рис. 16.12.

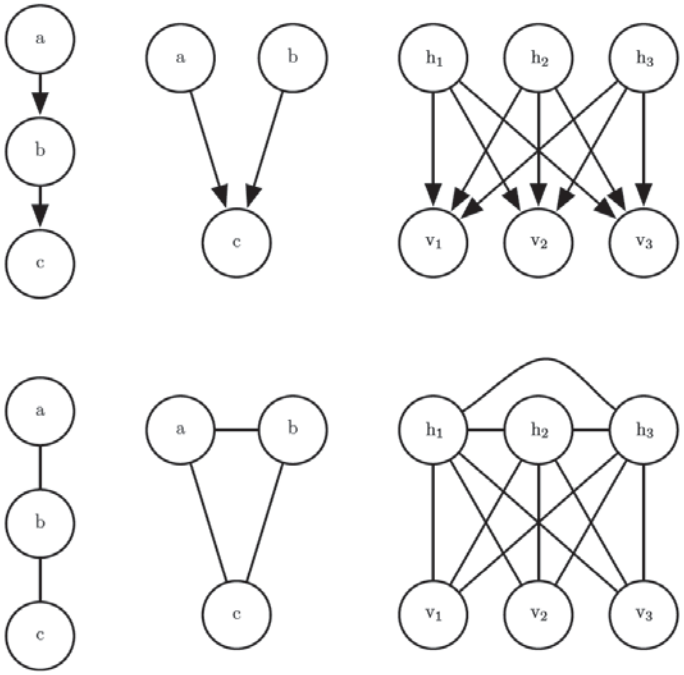

Рис. 16.11 ❖ Примеры преобразования ориентированных моделей (верхний ряд) в неориентированные (нижний ряд). (*Слева*) Эту простую цепочку можно преобразовать в моральный граф, просто заменив ориентированные ребра неориентированными. Из получившейся неориентированной модели вытекает точно такое же множество независимостей и условных независимостей. (*В центре*) Это простейшая ориентированная модель, которую невозможно преобразовать в неориентированную без потери части независимостей. Граф представляет собой одну-единственную аморальность. Поскольку a и b являются родителями c, то они соединены активным путем, если c наблюдаемая. Чтобы отразить эту зависимость, неориентированная модель должна включать клику, содержащую все три величины. Но эта клика не может представить тот факт, что $a \perp b$. (*Справа*) В общем случае морализация может добавить в граф много ребер, что приведет к утрате многих вытекающих из графа независимостей. Например, для этого графа разреженного кодирования нужно добавить морализирующие ребра между каждой парой скрытых блоков, что ведет к появлению квадратичного числа новых прямых зависимостей

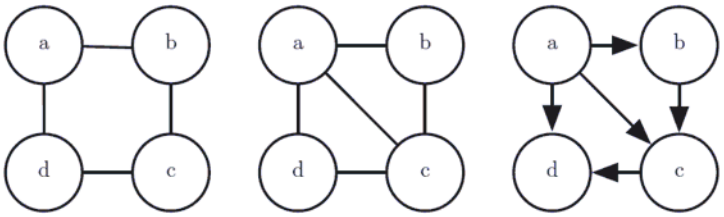

Рис. 16.12 ❖ Преобразование неориентированной модели в ориентированную. (*Слева*) Эту неориентированную модель нельзя преобразовать в ориентированную, потому что в ней есть цикл длины 4 без хорд. Это означает, что данная модель кодирует две разные независимости, которые никакая ориентированная модель не может представить одновременно: $a \perp c \mid \{b, d\}$ и $b \perp d \mid \{a, c\}$. (*В центре*) Для преобразования неориентированной модели в ориентированную необходимо триангулировать граф, так чтобы все циклы длины больше 3 содержали хорду. Для этого мы можем добавить ребро, соединяющее a и c, или ребро, соединяющее b и d. В этом примере мы добавили ребро между a и c. (*Справа*) Чтобы завершить процесс преобразования, необходимо назначить каждому ребру направление, не создав при этом ориентированных циклов. Для этого мы можем ввести отношение порядка на множестве вершин и назначать ребру направление так, чтобы меньшая вершина была начальной, а большая – конечной. В данном случае вершины упорядочены в алфавитном порядке своих имен

16.2.7. Факторные графы

Факторные графы – еще один способ изобразить неориентированную модель, устраняющий неоднозначность, присутствующую в стандартном синтаксисе. В неориентированной модели областью действия каждой функции ϕ должно быть *подмножество* некоторой клики графа. Неоднозначность возникает из-за того, что не понятно, действительно ли у каждой клики имеется фактор, область действия которого охватывает всю клику. Например, клика, содержащая три вершины, может соответствовать фактору над всеми тремя вершинами или трем факторам над каждой парой вершин. Факторные графы разрешают эту неоднозначность посредством явного представления области действия каждой функции ϕ. Точнее говоря, факторный граф – это графическое представление неориентированной модели, состоящее из двудольного неориентированного графа. Некоторые вершины изображаются кружочками – они соответствуют случайным величинам, как в стандартной неориентированной модели. Остальные вершины изображаются квадратиками и соответствуют факторам ϕ ненормированного распределения вероятности. Случайные величины и факторы могут быть соединены неориентированными ребрами. Величина и фактор соединены ребром тогда и только тогда, когда данная величина является одним из аргументов фактора в ненормированном распределении вероятности. Никакой фактор не может быть соединен ни с каким другим фактором, и никакая величина не может быть соединена с другой величиной. На рис. 16.13 приведен пример того, как факторные графы разрешают неоднозначность интерпретации неориентированных сетей

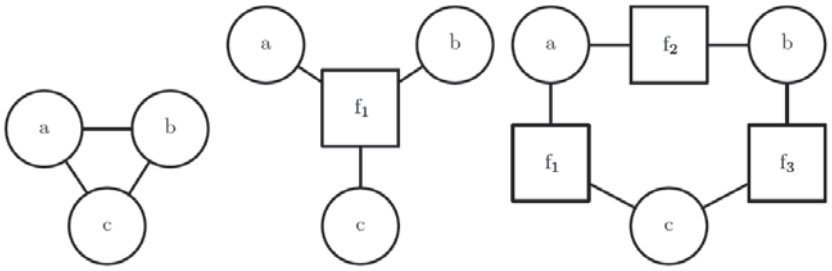

Рис. 16.13 ❖ Пример разрешения неоднозначности интерпретации неориентированных сетей с помощью факторного графа. (*Слева*) Неориентированная сеть с кликой из трех случайных величин a, b и c. (*В центре*) Факторный граф, соответствующий той же неориентированной модели. В этом графе имеется всего один фактор над всеми тремя величинами. (*Справа*) Другой допустимый факторный граф для той же модели – с тремя факторами, по одному для каждой пары величин. Представление, вывод и обучение для этого графа асимптотически дешевле, чем для графа в центре, несмотря на то что оба представляют одну и ту же неориентированную модель

16.3. Выборка из графических моделей

Графические модели также упрощают операцию выборки из модели. Одно из преимуществ ориентированных моделей состоит в том, что простая и эффективная процедура **предковой выборки** (ancestral sampling) может дать пример из совместного распределения, представленного моделью.

Основная идея заключается в том, чтобы произвести топологическую сортировку вершин x_i графа, так что для всех i и j справедливо утверждение: j больше i, если x_i – родитель x_j. Тогда выборку из случайных величин можно производить в этом порядке. Иначе говоря, сначала производим выборку $\mathrm{x}_1 \sim P(\mathrm{x}_1)$, затем из $P(\mathrm{x}_2 \mid Pa_{\mathcal{G}}(\mathrm{x}_2))$ и т. д., пока не дойдем до $P(\mathrm{x}_n \mid Pa_{\mathcal{G}}(\mathrm{x}_n))$. Если выборку из каждого условного распределения $P(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i))$ произвести легко, то легко сделать выборку и из всей модели. Топологическая сортировка гарантирует, что мы сможем прочитать условные распределения (16.1) и произвести из них выборку по порядку. Без сортировки мы могли бы попытаться произвести выборку случайной величины до того, как станут доступны ее родители.

Для некоторых графов топологический порядок не единственный. Предковую выборку можно применять с любым возможным порядком.

В общем случае предковая выборка работает очень быстро (в предположении, что производить выборку из каждого условного распределения легко) и удобно.

Недостаток предковой выборки – в том, что она применима только к ориентированным графическим моделям. Другой недостаток – в том, что предковая выборка не поддерживает произвольную операцию условной выборки. Если мы хотим произвести выборку из подмножества случайных величин в ориентированной графической модели при условии некоторых других величин, то часто требуется, чтобы все обусловливающие величины встречались в упорядоченном графе раньше, чем величины, из которых производится выборка. В таком случае мы можем выбрать из локальных условных распределений вероятности, заданных модельным распределением. В противном случае условные распределения, из которых мы должны произ-

водить выборку, являются апостериорными распределениями при условии наблюдаемых переменных. Эти апостериорные распределения обычно не заданы и не параметризованы в модели явно. Их вывод может оказаться дорогостоящей операцией. В тех моделях, где дело обстоит именно так, предковая выборка неэффективна.

К сожалению, предковая выборка применима только к ориентированным моделям. Для выборки из неориентированной модели мы можем сначала преобразовать ее в ориентированную, но при этом зачастую возникают неразрешимые задачи вывода (чтобы определить маргинальное распределение корневых вершин нового ориентированного графа) или приходится добавлять так много ребер, что получающаяся ориентированная модель становится вычислительно неприступной. Выборка из неориентированной модели без предварительного преобразования в ориентированную требует разрешения циклических зависимостей. Каждая величина взаимодействует со всеми другими величинами, поэтому у процесса выборки нет очевидной начальной точки. К сожалению, выборка из неориентированной графической модели является дорогостоящей многопроходной процедурой. Концептуально самым простым подходом является **выборка по Гиббсу**. Предположим, что имеется графическая модель над n-мерным вектором случайных величин $\mathbf{x}$. Мы итеративно посещаем каждую величину x_i и производим выборку, обусловленную всеми остальными величинами, из распределения $p(\mathrm{x}_i \mid \mathrm{x}_{-i})$. Благодаря свойствам разделенности графической модели это то же самое, что обусловливание только соседями x_i. К сожалению, после того как выполнен один проход по графической модели и произведена выборка из всех n величин, мы еще не получаем справедливой выборки $p(\mathbf{x})$. Мы должны повторить весь процесс, произведя повторную выборку из всех n величин, пользуясь обновленными значениями соседей. Асимптотически, после многих итераций, процесс сходится к выборке из корректного распределения. Но бывает трудно решить, когда достигнуто достаточно точное приближение к желаемому распределению. Методы выборки из неориентированных моделей – трудный и интересный вопрос, который более подробно рассматривается в главе 17.

16.4. Преимущества структурного моделирования

Основное преимущество структурных вероятностных моделей заключается в том, что они кардинально снижают стоимость представления распределений вероятности, а также обучения и вывода. В случае ориентированных моделей ускоряется еще и выборка, хотя для неориентированных моделей ситуация сложнее. Главная причина, из-за которой все эти операции требуют меньше времени и памяти, состоит в том, что некоторые взаимодействия не моделируются. Графические модели сообщают важную информацию посредством выбрасывания ребер. Если между какими-то величинами ребра нет, значит, их прямое взаимодействие моделировать не нужно.

У структурных вероятностных моделей есть также достоинство, в меньшей степени поддающееся количественной оценке: они позволяют явно отделить представление знаний от обучения знаниям или вывода с использованием существующих знаний. Поэтому такие модели проще разрабатывать и отлаживать. Мы можем проектировать, анализировать и оценивать алгоритмы обучения и вывода, применимые к широким классам графов. Заодно открывается возможность проектировать модели, отражающие те связи внутри данных, которые мы считаем важными. Комбинируя различные алгоритмы и структуры, мы можем получить декартово произведение раз-

нообразных возможностей. Было бы гораздо труднее отдельно проектировать комплексные алгоритмы для каждой возможной ситуации.

16.5. Обучение и зависимости

Хорошая порождающая модель должна верно отражать распределение наблюдаемых, или «видимых», переменных **v**. Часто различные элементы **v** сильно зависят друг от друга. В контексте глубокого обучения наиболее распространенный подход к моделированию таких зависимостей состоит в том, чтобы завести несколько латентных, или «скрытых», переменных **h**. Тогда модель может представить зависимости между любой парой переменных v_i и v_j косвенно, посредством прямых зависимостей между v_i и **h** и прямых зависимостей между **h** и v_j.

В хорошей модели **v**, не содержащей никаких латентных переменных, должно было бы присутствовать очень много родителей каждого узла в байесовской сети или очень большие клики в марковской сети. Даже просто представить такие взаимодействия высшего порядка обходится дорого – как в вычислительном смысле, поскольку число параметров, хранящихся в памяти, экспоненциально возрастает с ростом числа членов клики, так и в статистическом, т. к. при таком гигантском числе параметров для вычисления точной оценки нужно очень много данных.

Если модель предназначена для улавливания зависимостей между видимыми переменными, соединенными напрямую, то соединять между собой все переменные практически бессмысленно, поэтому граф следует проектировать так, чтобы были соединены лишь тесно связанные переменные, и опускать все остальные ребра. Этой проблеме посвящен целый раздел машинного обучения – **структурное обучение**. Хорошим справочным пособием по этой теме может служить книга (Koller and Friedman, 2009). По большей части методы структурного обучения сводятся к тому или иному виду жадного поиска. Предлагается некоторая структура, и обучается модель с такой структурой, затем ей присваивается оценка. Эта оценка поощряет за верность аппроксимации обучающего набора и штрафует за сложность модели. На следующем шаге поиска предлагаются модели-кандидаты с малым числом добавленных или удаленных ребер. Производится поиск новой структуры, от которой ожидается улучшение оценки.

Применение латентных переменных вместо адаптивной структуры позволяет отказаться от дискретных операций поиска и нескольких раундов обучения. Фиксированная структура над видимыми и латентными переменными может использовать прямые взаимодействия между видимыми и скрытыми блоками для определения косвенных взаимодействий между видимыми блоками. Применяя простые методы обучения параметров, мы можем обучить модель с фиксированной структурой, которая накладывает правильную структуру на маргинальное распределение $p(\mathbf{v})$.

Достоинства латентных переменных не исчерпываются их ролью в эффективном отражении $p(\mathbf{v})$. Новые переменные **h** дают также альтернативное представление **v**. Например, как было отмечено в разделе 3.9.6, модель гауссовой смеси обучается латентной переменной, соответствующей категории примеров, из которой выбирались входные данные. Это означает, что модель гауссовой смеси можно использовать для классификации. В главе 14 мы видели, что простые вероятностные модели типа разреженного кодирования обучаются латентным переменным, которые можно использовать как входные признаки для классификатора или как координаты на многооб-

разии. Так же можно использовать и другие модели, но глубокие модели и модели с разными видами взаимодействий способны создать еще более развитые описания входных данных. Во многих подходах обучение признаков достигается путем обучения латентным переменным. Часто при наличии некоторой модели $\mathbf{v}$ и $\mathbf{h}$ экспериментальные наблюдения показывают, что $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$ или $\arg\max_{\boldsymbol{h}} p(\boldsymbol{h}, \boldsymbol{v})$ – хорошее отображение признаков для $\boldsymbol{v}$.

16.6. Вывод и приближенный вывод

Один из основных способов использования вероятностной модели заключается в задавании вопросов о связи между различными величинами. Располагая набором медицинских анализов, мы можем спросить, чем болен пациент. В модели с латентными переменными интересно выделить признаки $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$, описывающие наблюдаемые переменные $\mathbf{v}$. Иногда подобные проблемы приходится решать для выполнения других задач. Часто мы обучаем модели, применяя принцип максимального правдоподобия. Поскольку

$$\log p(\boldsymbol{v}) = \mathbb{E}_{\mathbf{h} \sim p(\mathbf{h} \mid \mathbf{v})} [\log p(\boldsymbol{h}, \boldsymbol{v}) - \log p(\boldsymbol{h} \mid \boldsymbol{v})], \tag{16.9}$$

то требуется вычислить $p(\mathbf{h} \mid \boldsymbol{v})$, чтобы реализовать правило обучения. Все это примеры проблем **вывода**, в которых нужно предсказать значения одних переменных по значениям других или предсказать распределение вероятности одних переменных, зная значения других.

К сожалению, для большинства интересных глубоких моделей эти проблемы неразрешимы, даже если для их упрощения использовать структурную графическую модель. Граф позволяет представить сложные распределения высокой размерности с разумным числом параметров, но применяемые в глубоком обучении графы обычно недостаточно ограничительны, чтобы еще и обеспечить эффективный вывод.

Легко видеть, что вычисление маргинальной вероятности общей графической модели #P – трудная задача. Класс сложности #P – обобщение класса сложности NP. Для класса NP нужно только определить, есть ли у задачи решение, и, если есть, найти какое-нибудь. В случае класса #P требуется подсчитать число решений. Для построения графической модели в худшем случае рассмотрим определение модели над бинарными переменными в задаче выполнимости булевых формул в k-конъюнктивной нормальной форме (3-SAT). Мы можем считать, что эти переменные равномерно распределены, а затем добавить по одной бинарной латентной переменной на каждый дизъюнкт, которая показывает, выполняется ли этот дизъюнкт. Далее добавляется еще одна латентная переменная, показывающая, выполняются ли все дизюнкты. Это можно сделать, не создавая большую клику, путем построения дерева редукции латентных переменных, в котором каждый узел дерева сообщает, выполнены ди две другие переменные. Листьями этого дерева являются переменные для каждого дизъюнкта. Тогда корень дерева сообщает, выполнена ли вся формула. Вследствие равномерного распределения литералов маргинальное распределение корня дерева редукции определяет долю комбинаций значений переменных, решающих задачу. Это искусственный пример худшего случая, но NP-трудные графы регулярно возникают в практических ситуациях.

Следовательно, необходим приближенный вывод. В контексте глубокого обучения под этим обычно понимают вариационный вывод, когда истинное распределение

$p(\mathbf{h} \mid \boldsymbol{v})$ аппроксимируется максимально близким к нему распределением $q(\mathbf{h} \mid \mathbf{v})$. Эта и другие техники более подробно описаны в главе 19.

16.7. Подход глубокого обучения к структурным вероятностным моделям

Специалисты по глубокому обучению на практике пользуются тем же базовым вычислительным инструментарием, что и другие специалисты по машинному обучению, работающие со структурными вероятностными моделями. Но мы обычно по-другому комбинируем инструменты, так что получающиеся алгоритмы и модели сильно отличаются от традиционных графических моделей.

В глубоком обучении не всегда применяются очень уж глубокие графические модели. Глубина графической модели определяется в терминах ее графа, а не графа вычислений. Будем говорить, что глубина латентной переменной h_i равна j, если кратчайший путь от h_i к наблюдаемой переменной состоит из j шагов. Глубиной модели называется максимальная глубина по всем таким переменным h_i. Так определенная глубина отличается от глубины, индуцированной графом вычислений. Во многих порождающих моделях, встречающихся в глубоком обучении, латентных переменных нет вообще или имеется всего один слой таких переменных, но при этом используются графы вычислений для определения условных распределений внутри модели.

По существу, в глубоком обучении всегда присутствует идея распределенных представлений. Даже мелкие модели, применяемые для целей глубокого обучения (например, предобучение мелких моделей, из которых впоследствии будет составлена глубокая), почти всегда содержат один большой слой латентных переменных. В моделях глубокого обучения латентных переменных, как правило, больше, чем наблюдаемых. Сложные нелинейные взаимодействия между переменными имеют вид непрямых соединений, включающих несколько латентных переменных.

Напротив, в традиционных графических моделях большинство переменных хотя бы изредка наблюдается, даже если многие из них отсутствуют в некоторых обучающих примерах. В традиционных моделях по большей части используются члены высшего порядка и техника структурного обучения, чтобы выявить сложные нелинейные взаимодействия между переменными. Если латентные переменные и есть, то их обычно немного.

Методы проектирования латентных переменных в глубоком обучении также отличаются. Обычно проектировщик не стремится заранее придать латентным переменным какую-то определенную семантику – алгоритм обучения свободен придумывать любые концепции, необходимые для моделирования конкретного набора данных. В большинстве случае человеку нелегко интерпретировать латентные переменные по завершении обучения, хотя существуют методы визуализации, позволяющие хотя бы примерно понять, что именно они представляют. Когда латентные переменные используются в традиционных графических моделях, им часто приписывается вполне определенная семантика – тема документа, интеллектуальный уровень студента, болезнь, вызвавшая у пациента наблюдаемые симптомы, и т. д. Такие модели гораздо проще для интерпретации и зачастую имеют больше теоретических гарантий, но они хуже масштабируются на сложные задачи и, в отличие от глубоких моделей, не допускают повторного использования в различных контекстах.

Еще одно очевидное различие связано с типом связности, который обычно встречается в глубоком обучении. В типичной глубокой графической модели имеются большие группы блоков, каждый из которых связан с другими группами, так что взаимодействия между двумя группами можно описать одной матрицей. В традиционных графических моделях связей очень мало, и выбор связей для каждой переменной часто проектируется вручную. Проектирование структуры модели тесно связано с выбором алгоритма вывода. В традиционных подходах целью обычно является практическая реализуемость точного вывода. Если это ограничение слишком сильное, применяется популярный алгоритм приближенного вывода – **циклическое распространение доверия**. Оба подхода часто хорошо работают с разреженными графами. Для сравнения – в моделях, применяемых в глубоком обучении, каждый видимый блок v_i обычно связан со многими скрытыми блоками h_j, так чтобы **h** могло служить распределенным представлением v_i (и, быть может, еще нескольких наблюдаемых переменных). У распределенных представлений много достоинств, но с точки зрения графических моделей и вычислительной сложности у них есть недостаток – они обычно приводят к графам, недостаточно разреженным для применения традиционных методов вывода и циклического распространения доверия. Поэтому одно из самых разительных отличий между графическими моделями вообще и глубокими графическими моделями состоит в том, что метод циклического распространения доверия почти никогда не применяется для глубокого обучения. Вместо этого глубокие модели проектируются так, чтобы были эффективны алгоритмы выборки по Гиббсу или вариационного вывода. Еще следует принять во внимание, что поскольку модели глубокого обучения содержат очень много латентных переменных, на первый план выходит эффективность численных методов. Это дополнительный (помимо выбора высокоуровневого алгоритма вывода) довод в пользу группировки блоков в слои с помощью матрицы, описывающей взаимодействие между двумя слоями. Это позволяет реализовать отдельные шаги алгоритма посредством эффективных операций умножения матриц или их обобщений на разреженные графы, например перемножения блочно-диагональных матриц или свертки.

Наконец, для глубокого подхода к графическому моделированию характерна откровенная терпимость к неизвестному. Вместо того чтобы упрощать модель до тех пор, пока не появится возможность точно вычислить все интересующие нас величины, мы увеличиваем мощность модели до такой степени, что ее едва-едва можно обучить или использовать. Часто мы используем модели, маргинальные распределения которых вычислить невозможно, и довольствуемся возможностью произвести из них приближенную выборку. Встречаются модели с неразрешимой целевой функцией, которую даже аппроксимировать за разумное время невозможно, но тем не менее мы умудряемся приближенно обучить ее, если удается эффективно оценить градиент такой функции. Подход глубокого обучения зачастую состоит в том, чтобы понять, какой минимум информации абсолютно необходим, а затем придумать, как в кратчайшие сроки получить разумную аппроксимацию этой информации.

16.7.1. Пример: ограниченная машина Больцмана

Ограниченная машина Больцмана (ОМБ) (Smolensky, 1986), или **гармониум**, – типичный пример применения графических моделей в глубоком обучении. Сама по себе ОМБ не является глубокой моделью. У нее есть один слой латентных перемен-

ных, который можно использовать для представления входа. В главе 20 мы увидим, как можно использовать ОМБ для построения различных более глубоких моделей. А сейчас покажем, как ОМБ на практике воплощает многие идеи, применяемые в разнообразных глубоких графических моделях: ее блоки организованы в большие группы, называемые слоями, связи между слоями описываются матрицей, связность относительно плотная, при проектировании модели учтена возможность выборки по Гиббсу, и акцент сделан на том, чтобы предоставить алгоритму обучения свободу выявлять латентные переменные, семантика которых не была задана проектировщиком. В разделе 20.2 мы еще вернемся к ОМБ.

Каноническая ОМБ – это энергетическая модель с бинарными видимыми и скрытыми блоками. Ее функция энергии имеет вид

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h} - \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}, \tag{16.10}$$

где $\boldsymbol{b}$, $\boldsymbol{c}$ и $\boldsymbol{W}$ – вещественные подлежащие обучению параметры, на которые не накладываются ограничения. Видно, что модель разбита на две группы блоков: $\boldsymbol{v}$ и $\boldsymbol{h}$ – и что взаимодействие между ними описывается матрицей $\boldsymbol{W}$. Графически модель показана на рис. 16.14. Как ясно по рисунку, важным аспектом модели является отсутствие прямых взаимодействий между любыми двумя видимыми или любыми двумя скрытыми блоками (отсюда и слово «ограниченная»; в произвольной машине Больцмана допустимы любые связи).

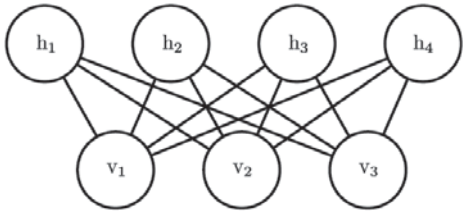

Рис. 16.14 ❖ ОМБ в виде марковской сети

Из ограничений на структуру ОМБ вытекают полезные свойства:

$$p(\mathbf{h} \mid \mathbf{v}) = \prod_i p(\mathrm{h}_i \mid \mathbf{v}) \tag{16.11}$$

и

$$p(\mathbf{v} \mid \mathbf{h}) = \prod_i p(\mathrm{v}_i \mid \mathbf{h}). \tag{16.12}$$

Отдельные условные вероятности также легко вычислить. Для бинарной ОМБ получаем:

$$P(\mathrm{h}_i = 1 \mid \mathbf{v}) = \sigma(\mathbf{v}^\top \boldsymbol{W}_{:,i} + b_i), \tag{16.13}$$

$$P(\mathrm{h}_i = 0 \mid \mathbf{v}) = 1 - \sigma(\mathbf{v}^\top \boldsymbol{W}_{:,i} + b_i). \tag{16.14}$$

В совокупности эти свойства позволяют эффективно производить **блочную выборку по Гиббсу**, когда выборка сразу всего $\mathbf{h}$ чередуется с выборкой сразу всего $\mathbf{v}$. Примеры, полученные в результате выборки по Гиббсу из модели ОМБ, показаны на рис. 16.15.

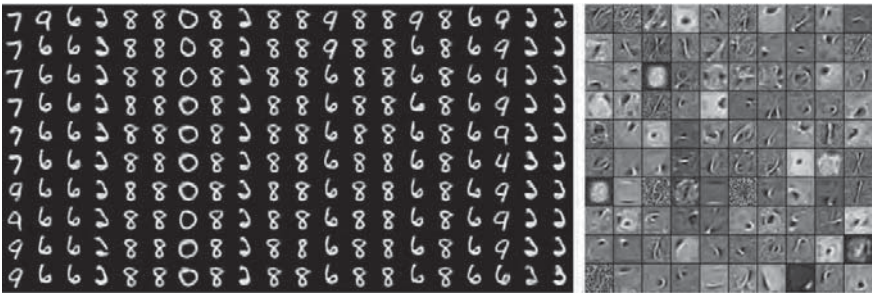

Рис. 16.15 ❖ Выборка из обученной ОМБ вместе с весами. (*Слева*) Примеры из модели, обученной на наборе данных MNIST, полученные с помощью выборки по Гиббсу. Каждый столбец – результат независимого процесса выборки. В каждой строке представлены результаты еще 1000 шагов выборки по Гиббсу. Соседние примеры сильно коррелированы между собой. (*Справа*) Соответствующие векторы весов. Сравните с примерами и весами для линейной факторной модели на рис. 13.2. Здесь примеры гораздо лучше, потому априорное распределение $p(\boldsymbol{h})$ не обязано быть факторным. В процессе выборки ОМБ может обучиться тому, какие признаки должны присутствовать вместе. С другой стороны, апостериорное распределение ОМБ $p(\boldsymbol{h} \mid \boldsymbol{v})$ факторное, тогда как апостериорное распределение разреженного кодирования таковым не является, поэтому модель разреженного кодирования может оказаться лучше для выделения признаков. Существуют и другие модели, в которых оба распределения $p(\boldsymbol{h})$ и $p(\boldsymbol{h} \mid \boldsymbol{v})$ нефакторные. Изображение взято из работы LISA (2008)

Поскольку функция энергии линейно зависит от параметров, ее производные легко вычисляются, например:

$$\frac{\partial}{\partial W_{i,j}} E(\mathbf{v}, \mathbf{h}) = -\mathrm{v}_i \mathrm{h}_j. \tag{16.15}$$

Эти два свойства: эффективная выборка по Гиббсу и эффективное вычисление производных – способствуют удобству обучения. В главе 18 мы увидим, что неориентированные модели можно обучать посредством вычисления таких производных для выборки из модели.

Обучение модели индуцирует представление $\boldsymbol{h}$ данных $\boldsymbol{v}$. Часто можно использовать $\mathbb{E}_{\mathbf{h} \sim p(\mathbf{h}|\boldsymbol{v})}[\boldsymbol{h}]$ как набор признаков для описания $\boldsymbol{v}$.

В целом ОМБ демонстрирует типичный подход глубокого обучения к графическим моделям: обучение представления в виде слоев латентных переменных в сочетании с эффективным взаимодействием между слоями, параметризованными с помощью матриц.

Графические модели предлагают элегантный, гибкий и ясный язык для описания вероятностных моделей. В следующих главах мы воспользуемся им (наряду с другими идеями) для описания широкого круга глубоких вероятностных моделей.

Глава 17

Методы Монте-Карло

Рандомизированные алгоритмы можно отнести к двум большим категориям: алгоритмы Лас-Вегаса и алгоритмы Монте-Карло. Алгоритм Лас-Вегаса всегда должен вернуть точный и правильный ответ (или сообщить об ошибке). Такие алгоритмы потребляют случайный объем ресурсов, обычно памяти или времени. Напротив, алгоритм Монте-Карло возвращает ответ со случайной ошибкой. Величину ошибки, как правило, можно уменьшить, увеличив потребление ресурсов (обычно времени работы или объема памяти). При любом фиксированном бюджете ресурсов алгоритм Монте-Карло может выдать приближенный ответ.

Многие задачи машинного обучения настолько трудны, что рассчитывать на получение точного ответа не приходится. Это сразу исключает точные детерминированные алгоритмы и алгоритмы Лас-Вегаса. Вместо них мы должны обходиться приближенными детерминированными алгоритмами или алгоритмами Монте-Карло. Оба подхода встречаются в машинном обучении повсеместно. В этой главе мы займемся методами Монте-Карло.

17.1. Выборка и методы Монте-Карло

Многие важные технологии, применяемые в машинном обучении, основаны на выборке примеров из некоторого распределения вероятности и их использовании для вычисления оценки интересующей величины по методу Монте-Карло.

17.1.1. Зачем нужна выборка?

Желание произвести выборку из распределения вероятности может возникнуть по разным причинам. Выборка – это гибкий способ относительно дешевой аппроксимации многих сумм и интегралов. Иногда мы применяем его, чтобы существенно ускорить хоть и осуществимое, но дорогостоящее вычисление; например, с помощью мини-пакетов мы уменьшаем полную стоимость обучения. В других случаях алгоритм обучения требует аппроксимировать недоступную для прямого вычисления сумму или интеграл, например градиент логарифма статистической суммы неориентированной модели. Бывает также, что выборка и есть конечная цель в том смысле, что мы хотим обучить модель, которая будет производить выборку из обучающего распределения.

17.1.2. Основы выборки методом Монте-Карло

Если сумму или интеграл нельзя вычислить точно (например, число слагаемых экспоненциально велико и, как упростить сумму, неизвестно), то часто можно аппроксими-

ровать ее с помощью выборки методом Монте-Карло. Идея в том, чтобы рассматривать сумму или интеграл как математическое ожидание относительно некоторого распределения и *аппроксимировать его с помощью соответствующего среднего*. Обозначим

$$s = \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f(\boldsymbol{x}) = E_p[f(\mathbf{x})] \tag{17.1}$$

или

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x} = E_p[f(\mathbf{x})] \tag{17.2}$$

подлежащую оценке сумму или интеграл, переписанные в виде математического ожидания, с тем ограничением, что p – распределение вероятности (в случае суммы) или плотность вероятности (в случае интеграла) случайной величины $\mathbf{x}$.

Мы можем аппроксимировать s, произведя выборку объема n из p: $\boldsymbol{x}^{(1)}$, ..., $\boldsymbol{x}^{(n)}$, а затем вычислив эмпирическое среднее:

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}^{(i)}). \tag{17.3}$$

В обоснование такой аппроксимации можно привести несколько соображений. Первое тривиальное наблюдение заключается в том, что оценка $\hat{s}$ несмещенная, т. к.

$$\mathbb{E}[\hat{s}_n] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}[f(\boldsymbol{x}^{(i)})] = \frac{1}{n} \sum_{i=1}^{n} s = s. \tag{17.4}$$

Кроме того, согласно **закону больших чисел**, если примеры $\boldsymbol{x}^{(i)}$ независимы и одинаково распределены, то средние почти наверняка сходятся к математическому ожиданию:

$$\lim_{n \to \infty} \hat{s}_n = s, \tag{17.5}$$

при условии что дисперсия отдельных членов $\mathrm{Var}[f(\boldsymbol{x}^{(i)})]$ ограничена. Чтобы убедиться в этом, рассмотрим дисперсию $\hat{s}_n$ с ростом n. Дисперсия $\mathrm{Var}[\hat{s}_n]$ убывает и сходится к 0, коль скоро $\mathrm{Var}[f(\mathbf{x}^{(i)})]] < \infty$:

$$\mathrm{Var}[\hat{s}_n] = \frac{1}{n^2} \sum_{i=1}^{n} \mathrm{Var}[f(\mathbf{x})], \tag{17.6}$$

$$= \frac{\mathrm{Var}[f(\mathbf{x})]}{n}. \tag{17.7}$$

Этот полезный результат заодно говорит, как оценить недостоверность среднего Монте-Карло, или, эквивалентно, величину ожидаемой погрешности аппроксимации Монте-Карло. Мы вычисляем одновременно эмпирическое среднее $f[(\boldsymbol{x}^{(i)})]$ и эмпирическую дисперсию[1], а затем делим оценку дисперсии на число примеров n для получения оценки $\mathrm{Var}[\hat{s}_n]$. Согласно **центральной предельной теореме**, распределение среднего $\hat{s}_n$ сходится к нормальному распределению со средним значением s и дисперсией $\mathrm{Var}[f(\boldsymbol{x})]/n$. Это позволяет оценить доверительные интервалы вокруг оценки $\hat{s}_n$ с помощью интегральной функции распределения нормальной плотности.

---

[1] Часто предпочтительнее несмещенная оценка дисперсии, в которой сумма квадратов разностей делится на $n - 1$, а не n.

Все это опирается на возможность произвести выборку из базового распределения $p(\mathbf{x})$, что, однако, не всегда возможно. Если выборка из p не осуществима, то можно вместо нее воспользоваться выборкой по значимости, описанной в разделе 17.2. Более общий подход – построить последовательность оценок, сходящуюся к интересующему нас распределению; он называется методом Монте-Карло по схеме марковских цепей (раздел 17.3).

17.2. Выборка по значимости

Важным шагом в декомпозиции подынтегрального выражения (или слагаемого) в выражении (17.2) является решение о том, какая его часть будет выступать в роли вероятности $p(\boldsymbol{x})$, а какая – в роли случайной величины $f(\boldsymbol{x})$, математическое ожидание которой (относительно данного распределения) мы хотим оценить. Не существует однозначно определенной декомпозиции, потому что $p(\boldsymbol{x})f(\boldsymbol{x})$ всегда можно переписать в виде

$$p(\boldsymbol{x})f(\boldsymbol{x}) = q(\boldsymbol{x})\frac{p(\boldsymbol{x})f(\boldsymbol{x})}{q(\boldsymbol{x})}, \tag{17.8}$$

так что выборка теперь производится из q, а оцениваемой величиной является pf/q. Во многих случаях мы хотим вычислить математическое ожидание для заданных p и f, и поскольку задача с самого начала поставлена как нахождение математического ожидания, то декомпозиция на p и f выглядит естественно. Однако исходная постановка задачи может быть неоптимальна с точки зрения количества примеров, необходимых для достижения заданной точности. По счастью, легко вывести, как выглядит оптимальный выбор q^*. Оптимальное q^* соответствует так называемой оптимальной выборке по значимости.

В силу тождества (17.8) любую оценку Монте-Карло

$$\hat{s}_p = \frac{1}{n}\sum_{i=1,\,\mathbf{x}^{(i)}\sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{17.9}$$

можно преобразовать в оценку выборки по значимости

$$\hat{s}_q = \frac{1}{n}\sum_{i=1,\,\mathbf{x}^{(i)}\sim q}^{n} \frac{p(\boldsymbol{x}^{(i)})f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})}. \tag{17.10}$$

Легко видеть, что математическое ожидание оценки не зависит от q:

$$\mathbb{E}_q[\hat{s}_q] = \mathbb{E}_q[\hat{s}_p] = s. \tag{17.11}$$

Однако дисперсия оценки выборки по значимости может быть весьма чувствительна к выбору q. Дисперсия вычисляется по формуле

$$\mathrm{Var}[\hat{s}_q] = \mathrm{Var}\left[\frac{p(\mathbf{x})f(\mathbf{x})}{q(\mathbf{x})}\right] \Big/ n. \tag{17.12}$$

Минимум дисперсии достигается, когда q равно

$$q^*(\boldsymbol{x}) = \frac{p(\boldsymbol{x})\left|f(\boldsymbol{x})\right|}{Z}, \tag{17.13}$$

где Z – нормировочная постоянная, выбранная так, чтобы сумма или интеграл $q^*(\boldsymbol{x})$ были равны 1. В лучших выборочных распределениях по значимости вес больше там, где больше подынтегральное выражение. На самом деле если $f(\boldsymbol{x})$ не меняет знак, то $\mathrm{Var}[\hat{s}_{q^*}] = 0$, т. е. при использовании оптимального распределения *достаточно одного примера*. Конечно, так происходит только потому, что вычисление q^* по существу уже решило исходную задачу, так что на практике не имеет особого смысла производить выборку одного примера из оптимального распределения.

Допустимо любое выборочное распределение q (в том смысле, что оно дает правильное значение математического ожидания), а q^* является оптимальным (в том смысле, что дает минимальную дисперсию). Выборка из q^* обычно неосуществима, но при другом выборе q может оказаться возможной, и при этом дисперсия все равно уменьшается.

Еще один подход состоит в использовании **смещенной выборки по значимости**, для которой не требуется нормировать p или q. Для дискретных случайных величин оценка смещенной выборки по значимости определяется по формуле

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})}} \tag{17.14}$$

$$= \frac{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{p(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{17.15}$$

$$= \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}}, \tag{17.16}$$

где $\tilde{p}$ и $\tilde{q}$ – ненормированные формы p и q, а $\boldsymbol{x}^{(i)}$ – выборка из q. Эта оценка смещена, потому что $\mathbb{E}[\hat{s}_{BIS}] \neq s$, а лишь асимптотически приближается к s, когда $n \to \infty$ и знаменатель в выражении (17.14) стремится к 1. Поэтому эта оценка называется асимптотически несмещенной.

Хороший выбор q может значительно повысить эффективность оценки методом Монте-Карло, но при плохом выборе эффективность может резко снизиться. Возвращаясь к формуле (17.12), мы видим, что если существуют выборочные примеры из q, для которых $(p(\boldsymbol{x})|f(\boldsymbol{x})|)/q(\boldsymbol{x})$ принимает большие значения, то дисперсия оценки будет очень велика. Так может случиться, если $q(\boldsymbol{x})$ близка к нулю, а $p(\boldsymbol{x})$ и $f(\boldsymbol{x})$ недостаточно малы, чтобы компенсировать это. Распределение q обычно выбирается простым, чтобы из него было легко произвести выборку. Когда размерность $\boldsymbol{x}$ велика, простое распределение q плохо аппроксимирует p или $p|f|$. Если $q(\boldsymbol{x}^{(i)}) \gg p(\boldsymbol{x}^{(i)})|f(\boldsymbol{x}^{(i)})|$, то выборка по значимости содержит бесполезные примеры (суммируются очень малые или нулевые значения). С другой стороны, если $q(\boldsymbol{x}^{(i)}) \ll p(\boldsymbol{x}^{(i)})|f(\boldsymbol{x}^{(i)})|$, что случается реже, то отношение будет очень велико. Поскольку такие события встречаются редко, в типичной выборке они могут отсутствовать, поэтому, как правило, мы получаем заниженную оценку s, которая изредка компенсируется сильно завышенной

оценкой. Такие очень большие или очень малые числа типичны при высокой размерности $\boldsymbol{x}$, поскольку в этом случае динамический диапазон совместных вероятностей может быть очень широк.

Несмотря на эту опасность, выборка по значимости и ее варианты оказались очень полезными во многих алгоритмах машинного обучения, в т. ч. глубокого. Например, она применяется для ускорения обучения в нейронных языковых моделях с большим словарем (раздел 12.4.3.3) и в других нейронных сетях с большим количеством выходов. См. также описание использования выборки по значимости для оценивания статистической суммы (нормировочной постоянной распределения вероятности) в разделе 18.7 и для оценивания логарифмического правдоподобия в таких глубоких моделях, как вариационный автокодировщик, – в разделе 20.10.3. Выборка по значимости полезна также для улучшения оценки градиента функции стоимости при обучении параметров модели методом стохастического градиентного спуска, особенно в моделях типа классификаторов, где основная часть величины функции стоимости приходится на небольшое число неправильно классифицированных примеров. В таких случаях более частая выборка более трудных примеров может уменьшить дисперсию градиента (Hinton, 2006).

17.3. Методы Монте-Карло по схеме марковской цепи

Во многих случаях мы хотели бы воспользоваться методом Монте-Карло, но невозможно произвести точную выборку из распределения $p_{\text{model}}(\mathbf{x})$ или из хорошего (с низкой дисперсией) выборочного распределения по значимости $q(\mathbf{x})$. В контексте глубокого обучения так чаще всего происходит, когда $p_{\text{model}}(\mathbf{x})$ представлено неориентированной моделью. В таких случаях применяется математический аппарат **марковских цепей** для приближенной выборки из $p_{\text{model}}(\mathbf{x})$. Семейство алгоритмов, в которых для получения оценки методом Монте-Карло используются марковские цепи, называется **методами Монте-Карло по схеме марковской цепи** (Markov chain Monte Carlo – MCMC). Применение MCMC-методов в машинном обучении подробно описано в книге Koller and Friedman (2009). Стандартные общие гарантии для MCMC-методов применимы, только когда модель не назначает ни одному состоянию нулевую вероятность. Поэтому их удобно представлять как выборку из энергетической модели $p(\boldsymbol{x}) \propto \exp(-E(\boldsymbol{x}))$, описанной в разделе 16.2.4, поскольку в этом случае гарантируется, что вероятности всех состояний ненулевые. На самом деле сфера применения MCMC-методов шире, их можно использовать и для многих распределений вероятности, включающих состояния с нулевой вероятностью. Однако теоретические гарантии относительно их поведения следует доказывать для каждого такого семейства распределений отдельно. В глубоком обучении принято полагаться на общие теоретические результаты, естественно распространяющиеся на все энергетические модели.

Чтобы понять, почему выборка из энергетической модели – трудное дело, рассмотрим энергетическую модель с двумя переменными, определяющую распределение $p(\mathrm{a}, \mathrm{b})$. Для выбора a мы должны произвести выборку из $p(\mathrm{a} \mid \mathrm{b})$, а для выбора b – из $p(\mathrm{b} \mid \mathrm{a})$. Получается неразрешимая проблема «яйцо или курица». В ориентированных моделях такой проблемы не возникает, потому что граф ориентированный и ацикли-

ческий. Чтобы произвести **предковую выборку**, мы просто выбираем значение каждой случайной величины в порядке топологической сортировки при условии ее родителей, для которых выборка уже гарантированно произведена (см. раздел 16.3). Предковая выборка определяет эффективный однопроходный метод получения выборки.

В энергетической модели порочный круг можно разорвать посредством выборки с применением марковской цепи. Основная идея марковской цепи – взять состояние $\boldsymbol{x}$, начинающееся с произвольного значения. С течением времени мы случайным образом изменяем $\boldsymbol{x}$. В конечном итоге $\boldsymbol{x}$ приближается к истинной выборке из $p(\boldsymbol{x})$. Формально говоря, марковская цепь определяется случайным состоянием $\boldsymbol{x}$ и переходным распределением $T(\boldsymbol{x}' \mid \boldsymbol{x})$, задающим вероятность того, что случайное изменение переведет состояние $\boldsymbol{x}$ в состояние $\boldsymbol{x}'$. Под выполнением марковской цепи понимается многократный переход из состояния $\boldsymbol{x}$ в состояние $\boldsymbol{x}'$ в соответствии с распределением $T(\mathbf{x}' \mid \boldsymbol{x})$.

Для теоретического осмысления принципов работы MCMC-методов полезно будет перепараметризовать задачу. Сначала сосредоточимся на случае, когда множество состояний случайной величины $\mathbf{x}$ счетно. Тогда любое состояние можно представить целым положительным числом x. Различные целые значения x можно затем отобразить на различные значения $\mathbf{x}$ в исходной задаче.

Посмотрим, что произойдет, если параллельно выполнять бесконечно много марковских цепей. Все состояния марковских цепей выбираются из некоторого распределения $q^{(t)}(x)$, где t – число уже произведенных шагов. В начальный момент $q^{(0)}$ – некоторое распределение, которое мы использовали для произвольной инициализации x для каждой марковской цепи. Затем на $q^{(t)}$ оказывают влияние все уже произведенные шаги марковской цепи. Наша цель – добиться, чтобы $q^{(t)}(x)$ сходилось к $p(x)$.

Поскольку мы перепараметризовали задачу в терминах целого положительного x, то можем описать распределение вероятности q с помощью вектора $\boldsymbol{v}$ – такого, что

$$q(\mathrm{x} = i) = v_i. \tag{17.17}$$

Посмотрим, что происходит, когда состояние x одной марковской цепи изменяется на x'. Вероятность, что новым состоянием будет x', равна

$$q^{(t+1)}(x') = \sum_x q^{(t)}(x) T(x' \mid x). \tag{17.18}$$

Пользуясь целочисленной параметризацией, мы можем представить действие оператора перехода T с помощью матрицы $\boldsymbol{A}$. Определим $\boldsymbol{A}$ следующим образом:

$$A_{i,j} = T(\mathbf{x}' = i \mid \mathbf{x} = j). \tag{17.19}$$

Перепишем формулу (17.18), воспользовавшись этим определением. Вместо того чтобы использовать q и T для описания изменения одного состояния, мы можем с помощью $\boldsymbol{v}$ и $\boldsymbol{A}$ описать, как меняется распределение всех марковских цепей (выполняемых параллельно) после перехода состояния:

$$\boldsymbol{v}^{(t)} = \boldsymbol{A}\boldsymbol{v}^{(t-1)}. \tag{17.20}$$

Изменение состояния марковской цепи соответствует умножению на матрицу $\boldsymbol{A}$. Иными словами, весь процесс можно описать как возведение матрицы $\boldsymbol{A}$ в степень:

$$\boldsymbol{v}^{(t)} = \boldsymbol{A}^t \boldsymbol{v}^{(0)}. \tag{17.21}$$

Матрица $\boldsymbol{A}$ обладает специальной структурой, поскольку ее столбцы представляют распределения вероятности. Такие матрицы называются **стохастическими**. Если существует ненулевая вероятность перехода из любого состояния x в любое другое состояние x' для некоторой степени t, то по теореме Перрона–Фробениуса (Perron, 1907; Frobenius, 1908) наибольшее собственное значение матрицы вещественно и равно 1. Мы видим, что с течением времени все собственные значения возводятся в степень:

$$\boldsymbol{v}^{(t)} = (\boldsymbol{V}\,\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1})^t\boldsymbol{v}^{(0)} = \boldsymbol{V}\,\mathrm{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}\boldsymbol{v}^{(0)}. \tag{17.22}$$

В результате все собственные значения, не равные 1, стремятся к нулю. При некоторых довольно мягких условиях гарантируется, что $\boldsymbol{A}$ имеет только один собственный вектор с собственным значением 1. Поэтому процесс сходится к **стационарному распределению**, которое иногда называют **равновесным распределением**. В пределе

$$\boldsymbol{v}' = \boldsymbol{A}\boldsymbol{v} = \boldsymbol{v}, \tag{17.23}$$

и это условие справедливо для каждого дополнительного шага. Это не что иное, как уравнение собственного вектора. Чтобы оказаться стационарной точкой, $\boldsymbol{v}$ должен быть собственным вектором с собственным значением 1. Это условие гарантирует, что после достижения стационарного распределения последующее применение процедуры переходной выборки не изменяет *распределения* состояний всех марковских цепей (хотя, разумеется, оператор перехода изменяет каждое отдельное состояние).

Если T выбрано правильно, то стационарное распределение q будет совпадать с распределением p, из которого мы хотим производить выборку. В разделе 17.4 мы расскажем, как выбирать T.

Большую часть свойств марковских цепей со счетным множеством состояний можно обобщить на непрерывные величины. В такой ситуации некоторые авторы называют марковскую цепь **цепью Харриса**, но мы будем в обоих случаях использовать термин «марковская цепь». В общем случае марковская цепь с оператором перехода T сходится (при довольно мягких условиях) к неподвижной точке, описываемой уравнением

$$\boldsymbol{q}'(\mathbf{x}') = \mathbb{E}_{\mathbf{x}\sim q}T(\mathbf{x}' \mid \mathbf{x}), \tag{17.24}$$

которое в дискретном случае сводится к уравнению (17.23). Если $\mathbf{x}$ – дискретная величина, то математическое ожидание записывается в виде суммы, а если непрерывная, то в виде интеграла.

Независимо от того, является состояние непрерывной или дискретной случайной величиной, все методы на основе марковских цепей заключаются в повторном применении стохастического обновления до тех пор, пока состояние не начнет давать выборку из равновесного распределения. Выполнение марковской цепи до достижения равновесного распределения называется **приработкой** марковской цепи. Затем из равновесного распределения можно выбирать бесконечно много примеров. Все они имеют одинаковое распределение, но любые два соседних примера сильно коррелированы между собой. Поэтому конечная последовательность примеров может оказаться недостаточно репрезентативной выборкой из равновесного распределения. Один из путей смягчения этой проблемы – возвращать каждый n-ый пример, чтобы оценка ста-

тистики равновесного распределения была в меньшей степени смещена из-за корреляции между каждым примером и несколькими следующими за ним. Использованию марковских цепей присущи высокие накладные расходы из-за времени приработки к равновесному распределению и времени перехода от одного примера к следующему, слабо коррелированному с ним, уже после достижения равновесия. Если необходимы действительно независимые примеры, то можно параллельно выполнять несколько марковских цепей. При таком подходе применяется распараллеливание вычислений, чтобы избежать задержки. Стратегия использования единственной марковской цепи для порождения всех примеров и стратегия использования отдельной цепи для каждого примера – две крайности; на практике в глубоком обучении количество цепей берут примерно равным числу примеров в мини-пакете, а затем выбирают столько примеров, сколько нужно, из этого фиксированного набора марковских цепей. Число цепей часто равно 100.

Еще одна трудность связана с тем, что мы заранее не знаем, сколько нужно выполнить шагов, прежде чем марковская цепь достигнет равновесного распределения. Этот промежуток времени иногда называют **временем приработки**, или **временем перемешивания** (mixing time). Проверить, достигла ли марковская цепь равновесия, тоже трудно. Теория недостаточно точна для ответа на этот вопрос. Утверждается лишь, что цепь сходится, но не более того. Анализ марковской цепи с точки зрения воздействия матрицы $\boldsymbol{A}$ на вектор вероятностей $\boldsymbol{v}$ показывает, что цепь приработалась, когда $\boldsymbol{A}^t$ потеряла практически все собственные значения $\boldsymbol{A}$, кроме единственного, равного 1. Это означает, что абсолютная величина второго по величине собственного значения определяет время перемешивания. Но на практике мы не можем представить марковскую цепь матрицей. Число возможных состояний вероятностной модели экспоненциально зависит от числа переменных, поэтому представить $\boldsymbol{v}$, $\boldsymbol{A}$ или собственные значения $\boldsymbol{A}$ не получится. Из-за этого и других препятствий мы обычно не знаем, приработалась ли цепь. Вместо этого мы просто даем цепи проработать какое-то время, которое считаем достаточным, исходя из грубой оценки, и применяем эвристические методы, чтобы понять, приработалась ли цепь. К числу таких методов относится просмотр примеров вручную или измерение корреляции между соседними примерами.

17.4. Выборка по Гиббсу

До сих пор мы говорили о том, как производить выборку из распределения $q(\boldsymbol{x})$ путем повторного обновления $\boldsymbol{x} \leftarrow \boldsymbol{x}' \sim T(\boldsymbol{x}' \mid \boldsymbol{x})$. Мы ничего не сказали о том, как убедиться, что $q(\boldsymbol{x})$ – полезное распределение. В этой книге рассматриваются два основных подхода. Первый – вывести T из заданного обученного распределения p_{model} – описан ниже вместе со случаем выборки из энергетической модели. Второй – непосредственно параметризовать и обучить T, так чтобы его стационарное распределение неявно определяло интересующее нас распределение p_{model}. Примеры второго подхода обсуждаются в разделах 20.12 и 20.13.

В контексте глубокого обучения мы обычно используем марковские цепи, чтобы производить выборку из энергетической модели, определяющей распределение $p_{\text{model}}(\boldsymbol{x})$. В этом случае мы хотим, чтобы распределение $q(\boldsymbol{x})$ для марковской цепи совпадало с $p_{\text{model}}(\boldsymbol{x})$. Для получения желаемого $q(\boldsymbol{x})$ необходимо выбрать подходящее распределение $T(\boldsymbol{x}' \mid \boldsymbol{x})$.

Концептуально простой и эффективной способ построения марковской цепи, которая производит выборку из $p_{\text{model}}(\boldsymbol{x})$, дает **выборка по Гиббсу**, когда выборка из $T(\mathbf{x}' \mid \mathbf{x})$ производится путем выбора одной величины x_i и выборки ее значений из p_{model} при условии соседей в неориентированном графе $\mathcal{G}$, определяющем структуру энергетической модели. Мы можем также одновременно производить выборку нескольких величин, если только они условно независимы при условии всех своих соседей. Как показано в примере ОМБ в разделе 16.7.1, из всех скрытых блоков можно производить выборку одновременно, потому что они условно независимы друг от друга при условии всех видимых блоков. И точно так же можно одновременно производить выборку из всех видимых блоков, потому что они условно независимы друг от друга при условии всех скрытых блоков. Если одновременно обновляется несколько величин, то говорят о **блочной выборке по Гиббсу**.

Есть и другие подходы к проектированию марковских цепей для выборки из p_{model}. Например, в других дисциплинах широко используется алгоритм Метрополиса–Гастингса. Но в глубоком обучении применительно к неориентированному моделированию редко применяется что-то, кроме выборки по Гиббсу. Улучшение методов выборки – передний край исследований.

17.5. Проблема перемешивания разделенных мод

Главная трудность, присущая МСМС-методам, – плохое **перемешивание**. В идеале последовательные примеры, выбранные из марковской цепи, спроектированной для выборки из $p(\boldsymbol{x})$, должны быть абсолютно независимы, и частота попадания в различные области пространства $\boldsymbol{x}$ должна быть пропорциональна вероятности области. На самом же деле, особенно в пространствах высокой размерности, примеры, выбранные МСМС-методом, оказываются сильно коррелированными. То есть налицо медленное перемешивание или даже полное отсутствие перемешивания. Можно считать, что МСМС-методы с медленным перемешиванием непреднамеренно выполняют нечто вроде зашумленного градиентного спуска вдоль функции энергии, или, эквивалентно, зашумленного восхождения на вершину функции вероятности относительно состояния цепи (отбираемых случайных величин). Цепь демонстрирует тенденцию к малым шагам (в пространстве состояний марковской цепи) из конфигурации $\boldsymbol{x}^{(t-1)}$ в конфигурацию $\boldsymbol{x}^{(t)}$, когда энергия $E(\boldsymbol{x}^{(t)})$ в общем случае меньше или приблизительно равна энергии $E(\boldsymbol{x}^{(t-1)})$, причем предпочтение отдается переходам в конфигурации с более низкой энергией. Начав с маловероятной конфигурации (энергия выше, чем для типичных примеров, выбранных из $p(\mathbf{x})$), цепь стремится постепенно уменьшать энергию состояния и лишь изредка переходит на другую моду. После того как цепь нашла область низкой энергии (например, в случае, когда случайными величинами являются пиксели изображения, областью низкой энергии может быть связное многообразие изображений одного и того же объекта), которую мы называем модой, она начинает перемещаться вокруг этой моды (совершая своего рода случайное блуждание). Время от времени цепь покидает эту моду и либо возвращается к ней, либо (если найдет путь выхода) переходит к другой моде. Проблема в том, что для многих интересных распределений успешные пути выхода встречаются редко, поэтому марковская цепь продолжает выбирать примеры из одной и той же моды дольше, чем хотелось бы.

Это очень наглядно проявляется при рассмотрении алгоритма выборки по Гиббсу (раздел 17.4). В этом контексте рассмотрим вероятность перехода из одной моды в со-

седнюю за данное количество шагов. Эта вероятность определяется формой «энергетического барьера» между модами. Переходы между модами, разделенными высоким барьером (областью низкой вероятности), экспоненциально менее вероятны (в терминах высоты барьера). Это показано на рис. 17.1. Проблема возникает, когда есть несколько мод с высокой вероятностью, разделенных областями низкой вероятности, особенно если каждый шаг выборки по Гиббсу должен обновить только небольшое подмножество переменных, значения которых в основном определяются другими переменными.

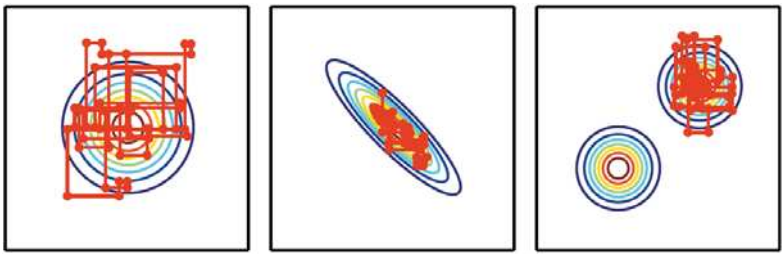

Рис. 17.1 ❖ Пути следования выборки по Гиббсу для трех распределений, во всех случаях марковская цепь инициализирована внутри моды. (*Слева*) Многомерное нормальное распределение двух независимых величин. Выборка по Гиббсу хорошо перемешивается, поскольку величины независимы. (*В центре*) Многомерное нормальное распределение сильно коррелированных величин. Из-за корреляции перемешивание марковской цепи затруднено. Поскольку обновление каждой величины должно быть обусловлено другой величиной, наличие корреляции замедляет скорость ухода марковской цепи от начальной точки. (*Справа*) Смесь нормальных распределений с широко разделенными модами, не находящимися на одной оси. Выборка по Гиббсу перемешивается очень медленно, потому что трудно сменить моду, изменяя в каждый момент времени только одну величину

В качестве простого примера рассмотрим энергетическую модель двух бинарных случайных величин a и b, принимающих значения -1 и 1. Если $E(\mathrm{a}, \mathrm{b}) = -w\mathrm{ab}$ для большого положительного числа w, то модель выражает сильную веру в то, что знаки a и b одинаковы. Рассмотрим обновление b посредством шага выборки по Гиббсу с $\mathrm{a} = 1$. Условное распределение b описывается формулой $P(\mathrm{b} = 1 \mid \mathrm{a} = 1) = \sigma(w)$. Если w велико, то сигмоида насыщается, и вероятность, что b тоже будет присвоено значение 1, близка к 1. Аналогично, если $\mathrm{a} = -1$, то вероятность, что и b будет равно -1, близка к 1. Согласно распределению $P_{\text{model}}(\mathrm{a}, \mathrm{b})$, знаки обеих величин равновероятны. Согласно же $P_{\text{model}}(\mathrm{a} \mid \mathrm{b})$, обе величины должны иметь одинаковый знак. Следовательно, выборка по Гиббсу очень редко изменяет знаки этих величин.

На практике проблема даже более серьезна, потому что нас интересуют не только переходы между двумя модами, но и вообще между всеми многочисленными модами, присутствующими в реальной модели. Если несколько таких переходов затруднено из-за сложности перемешивания мод, то получить надежный набор примеров, охватывающий большинство мод, будет чрезвычайно дорого, а сходимость цепи к стационарному распределению окажется очень медленной.

Иногда проблему можно решить путем нахождения групп сильно зависимых блоков и их одновременного обновления. К сожалению, когда зависимости сложны, выборка из группы становится вычислительно неразрешимой задачей. В конце концов, проблема, которую марковские цепи и призваны были решить, – это проблема выборки из большой группы случайных величин.

В контексте моделей с латентными переменными, которые определяют совместное распределение $p_{\text{model}}(\boldsymbol{x}, \boldsymbol{h})$, мы часто производим выборку из $\boldsymbol{x}$, чередуя выборку из $p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$ с выборкой из $p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x})$. С точки зрения скорости перемешивания, мы хотели бы, чтобы у $p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x})$ была высокая энтропия. А с точки зрения обучения полезного представления $\boldsymbol{h}$, нам нужно, чтобы в $\boldsymbol{h}$ было закодировано достаточно информации о $\boldsymbol{x}$ для ее успешной реконструкции. Это означает, что взаимная информация $\boldsymbol{h}$ и $\boldsymbol{x}$ должна быть велика. Эти две цели противоречат друг другу. Мы часто обучаем порождающие модели, которые очень точно кодируют $\boldsymbol{x}$ в $\boldsymbol{h}$, но плохо перемешиваются. Такая ситуация часто возникает в случае машин Больцмана – чем острее распределение, обучаемое машиной Больцмана, тем сложнее обеспечить хорошее перемешивание выборки из распределения модели с помощью марковской цепи. Эта проблема иллюстрируется на рис. 17.2.

Все это могло бы снизить полезность МСМС-методов, когда интересующее нас распределение имеет структуру нескольких многообразий, по одному для каждого класса: распределение концентрируется вокруг нескольких мод, разделенных обширными областями высокой энергии. Именно такого типа распределения мы ожидаем во многих задачах классификации, но тогда МСМС-методы сходились бы очень медленно из-за плохого перемешивания мод.

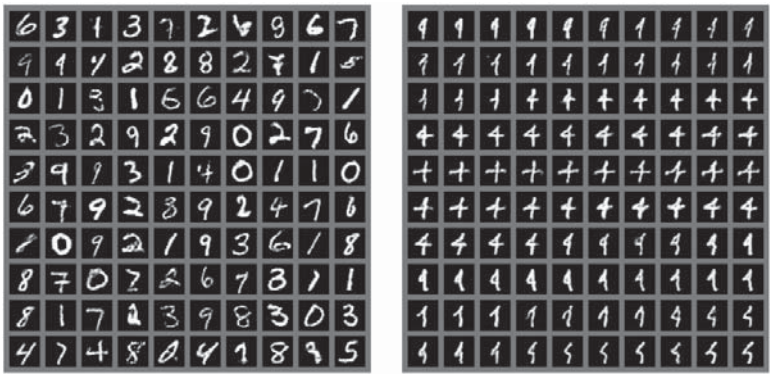

Рис. 17.2 ❖ Иллюстрация проблемы медленного перемешивания в глубоких вероятностных моделях. Каждую таблицу следует читать слева направо сверху вниз. (*Слева*) Соседние примеры из выборки по Гиббсу, примененной к глубокой машине Больцмана, обученной на наборе данных MNIST. Соседние примеры похожи друг на друга. Поскольку выборка по Гиббсу производится в глубокой графической модели, это сходство основано скорее на семантике, чем на визуальных признаках, но все равно марковской цепи трудно перейти из одной моды распределения в другую, например путем изменения цифры. (*Справа*) Соседние примеры для предковой выборки из порождающей состязательной сети. Поскольку предковая выборка порождает примеры независимо, то проблемы перемешивания нет

17.5.1. Применение темперирования для перемешивания мод

Если в распределении имеются острые пики высокой вероятности, окруженные областями низкой вероятности, то перемешивание разных мод затруднено. Несколько приемов повышения скорости перемешивания основано на построении альтернативных вариантов целевого распределения, в котором пики не такие высокие, а окружающие их долины не такие низкие. В энергетических моделях сделать это особенно просто. До сих пор мы описывали энергетическую модель как определяющую распределение вероятности вида

$$p(\boldsymbol{x}) \propto \exp(-E(\boldsymbol{x})). \tag{17.25}$$

Энергетические модели можно дополнить параметром β, контролирующим остроту пика:

$$p_\beta(\boldsymbol{x}) \propto \exp(-\beta E(\boldsymbol{x})). \tag{17.26}$$

Параметр β часто называют обратной **температурой**, что отражает истоки энергетических моделей – статистическую физику. Когда температура стремится к нулю, а β устремляется к бесконечности, энергетическая модель становится детерминированной. Если же температура стремится к бесконечности, а β – к нулю, то распределение (для дискретных $\boldsymbol{x}$) становится равномерным.

Обычно модель обучают при $\beta = 1$. Однако можно использовать и другие температуры, в частности $\beta < 1$. **Темперирование** (tempering)[1] – это общая стратегия быстрого перемешивания мод p_1 путем выборки примеров с $\beta < 1$.

Марковские цепи, основанные на **темперированных переходах** (Neal, 1994), временно производят выборку из высокотемпературных распределений, чтобы перемешать разные моды, а затем возобновляют выборку из распределения с единичной температурой. Такие методы применялись к моделям типа ОМБ (Salakhutdinov, 2010). Другой подход – использование **параллельного темперирования** (Iba, 2001), когда марковская цепь параллельно имитирует много различных состояний при разных температурах. Высокотемпературные состояния медленно перемешиваются, а низкотемпературные (с температурой 1) дают верные выборки из модели. Оператор перехода включает стохастический обмен состояний из двух разных температурных режимов, так чтобы пример с достаточно большой вероятностью из высокотемпературного состояния мог перескочить в низкотемпературное. Этот подход также применялся к ОМБ (Desjardins et al., 2010; Cho et al., 2010). Хотя темперирование – многообещающая идея, в настоящее время она не позволила далеко продвинуться в решении проблемы выборки из сложных энергетических моделей. Возможно, дело в том, что существуют **критические температуры**, в окрестности которых температурный переход (постепенное понижение температуры) должен быть очень медленным, и только тогда темперирование оказывается эффективным.

17.5.2. Глубина может помочь перемешиванию

Говоря о выборке из модели с латентными переменными $p(\boldsymbol{h}, \boldsymbol{x})$, мы видели, что если $p(\boldsymbol{h} \mid \boldsymbol{x})$ кодирует $\boldsymbol{x}$ слишком хорошо, то выборка из $p(\boldsymbol{x} \mid \boldsymbol{h})$ не сильно изменяет $\boldsymbol{x}$, и перемешивание оказывается плохим. Один из способов решения этой проблемы – сде-

[1] Темперированием в металлургии называют процесс закалки сплава с последующим отпуском. – *Прим. перев.*

лать $\boldsymbol{h}$ глубоким представлением, закодировав $\boldsymbol{x}$ в $\boldsymbol{h}$ таким образом, чтобы марковская цепь в пространстве $\boldsymbol{h}$ легче перемешивалась. Многие алгоритмы обучения представлений, в т. ч. автокодировщики и ОМБ, имеют тенденцию порождать маргинальное распределение $\boldsymbol{h}$, более равномерное и более близкое к унимодальному, чем исходное распределение $\boldsymbol{x}$. Можно предположить, что это объясняется попыткой минимизировать ошибку реконструкции, используя все доступное пространство представления, поскольку минимизацию ошибки реконструкции по обучающим примерам проще осуществить, когда различные примеры легко различимы в $\boldsymbol{h}$-пространстве и потому хорошо разделены. В работе Bengio et al. (2013a) отмечено, что более глубокие стеки регуляризированных автокодировщиков или ОМБ дают маргинальные распределения в $\boldsymbol{h}$-пространстве верхнего уровня, которые выглядят более вытянутыми и более равномерными, причем промежуток между областями, соответствующими разным модам (в экспериментах – категориям), меньше. Обучение ОМБ в этом пространстве более высокого уровня привело к более быстрому перемешиванию мод при выборке по Гиббсу. Однако остается неясным, как воспользоваться этим наблюдением в целях улучшения обучения и выборки из глубоких порождающих моделей.

Несмотря на трудности перемешивания, методы Монте-Карло полезны и зачастую являются лучшими из имеющихся инструментов. На самом деле это основной инструмент борьбы с вычислительно неразрешимой статистической суммой в неориентированных моделях. Именно об этом пойдет речь в следующей главе.

Глава 18

Преодоление трудностей, связанных со статической суммой

В разделе 16.2.2 мы видели, что многие вероятностные модели (точнее, неориентированные графические модели) определены в терминах ненормированного распределения вероятности $\tilde{p}(\mathbf{x}; \theta)$. Для получения корректного распределения вероятности мы должны нормировать $\tilde{p}$, поделив на статистическую сумму $Z(\boldsymbol{\theta})$:

$$p(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \tilde{p}(\mathbf{x}; \boldsymbol{\theta}). \tag{18.1}$$

Статистическая сумма – это интеграл (для непрерывных величин) или сумма (для дискретных величин) ненормированных вероятностей всех состояний:

$$\int \tilde{p}(\boldsymbol{x}) d\boldsymbol{x} \tag{18.2}$$

или

$$\sum_{x} \tilde{p}(\boldsymbol{x}). \tag{18.3}$$

Для многих интересных моделей эта операция вычислительно неразрешима.

В главе 20 мы увидим, что несколько моделей глубокого обучения специально спроектировано так, чтобы нормировочную постоянную можно было вычислить, или так, чтобы $p(\mathbf{x})$ можно было не вычислять вовсе. Тем не менее в других моделях приходится сталкиваться с проблемой неразрешимых статистических сумм. В этой главе мы опишем методы, применяемые для обучения и использования моделей с неразрешимыми статистическими суммами.

18.1. Градиент логарифмического правдоподобия

Обучение неориентированных моделей методом максимального правдоподобия особенно осложняется тем, что статистическая сумма зависит от параметров. Градиент логарифмического правдоподобия по параметрам содержит член, соответствующий градиенту статистической суммы:

$$\nabla_{\theta} \log p(\mathbf{x}; \boldsymbol{\theta}) = \nabla_{\theta} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) - \nabla_{\theta} \log Z(\boldsymbol{\theta}). \tag{18.4}$$

Это хорошо известное разложение на **положительную** и **отрицательную фазы** обучения.

Для большинства интересных неориентированных моделей отрицательная фаза представляет сложности. Если в модели нет латентных переменных или между ними немного взаимодействий, то положительная фаза обычно разрешима. Типичный пример модели с простой положительной и трудной отрицательной фазой – ОМБ, в которой имеются скрытые блоки, условно независимые друг от друга при условии видимых блоков. Случай трудной положительной фазы со сложными взаимодействиями между латентными переменными рассмотрен в главе 19. Здесь же мы займемся трудностями отрицательной фазы.

Присмотримся к градиенту $\log Z$ поближе:

$$\nabla_{\boldsymbol{\theta}} \log Z, \tag{18.5}$$

$$= \frac{\nabla_{\boldsymbol{\theta}} Z}{Z} \tag{18.6}$$

$$= \frac{\nabla_{\boldsymbol{\theta}} \sum_{\mathbf{x}} \tilde{p}(\mathbf{x})}{Z} \tag{18.7}$$

$$= \frac{\sum_{\mathbf{x}} \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})}{Z}. \tag{18.8}$$

Если модель гарантирует, что $p(\mathbf{x}) > 0$ для всех $\mathbf{x}$, то мы можем подставить $\exp(\log \tilde{p}(\mathbf{x}))$ вместо $\tilde{p}(\mathbf{x})$:

$$\frac{\sum_{\mathbf{x}} \nabla_{\boldsymbol{\theta}} \exp(\log \tilde{p}(\mathbf{x}))}{Z} \tag{18.9}$$

$$= \frac{\sum_{\mathbf{x}} \exp(\log \tilde{p}(\mathbf{x})) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x})}{Z} \tag{18.10}$$

$$= \frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x})}{Z} \tag{18.11}$$

$$= \sum_{\mathbf{x}} p(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}) \tag{18.12}$$

$$= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}). \tag{18.13}$$

В этом выводе используется суммирование по дискретной величине $\boldsymbol{x}$, но аналогичный результат имеет место для интегрирования по непрерывной $\boldsymbol{x}$. В этом случае мы применяем правило Лейбница дифференцирования под знаком интеграла:

$$\nabla_{\boldsymbol{\theta}} \int \tilde{p}(\mathbf{x}) d\boldsymbol{x} = \int \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x}) d\boldsymbol{x}. \tag{18.14}$$

Это тождество применимо только при некоторых условиях регулярности на $\tilde{p}$ и $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$. В терминах теории меры эти условия формулируются так: (1) ненормированное распределение $\tilde{p}$ должно быть интегрируемой по Лебегу функцией $\boldsymbol{x}$ для любого значения $\boldsymbol{\theta}$; (2) градиент $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ должен существовать для всех $\boldsymbol{\theta}$ и почти всех $\mathbf{x}$; (3) должна существовать интегрируемая функция $R(\mathbf{x})$, ограничивающая $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ в том смысле, что $\max_i |(\partial / \partial \theta_i) \tilde{p}(\mathbf{x})| \leq R(\boldsymbol{x})$ для всех $\boldsymbol{\theta}$ и почти всех $\boldsymbol{x}$. К счастью, большинство интересных моделей машинного обучения обладает этими свойствами.

Следующее тождество

$$\nabla_{\boldsymbol{\theta}} \log Z = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}) \tag{18.15}$$

лежит в основе разнообразных методов Монте-Карло для приближенной максимизации правдоподобия моделей с неразрешимыми статистическими суммами.

Подход к обучению неориентированных моделей методом Монте-Карло предлагает интуитивно понятную схему, в рамках которой мы можем рассуждать о положительной и отрицательной фазах. В положительной фазе мы увеличиваем $\log \tilde{p}(\mathbf{x})$ для $\boldsymbol{x}$, выбранного из данных. В отрицательной фазе мы уменьшаем статистическую сумму за счет уменьшения $\log \tilde{p}(\mathbf{x})$ для $\boldsymbol{x}$, выбранного из модельного распределения.

В литературе по глубокому обучению принято параметризовать $\log \tilde{p}$ в терминах функции энергии (уравнение 16.7). В этом случае мы можем интерпретировать положительную фазу как толкание вниз энергии обучающих примеров, а отрицательную фазу – как толкание вверх энергии примеров, выбранных из модели (см. рис. 18.1).

18.2. Стохастическая максимизация правдоподобия и сопоставительное расхождение

Наивный способ реализации уравнения (18.15) состоит в том, чтобы вычислить его посредством приработки множества марковских цепей, инициализированных случайным образом, всякий раз как понадобится градиент. Если обучение производится методом стохастического градиентного спуска, то это означает, что приработка должна производиться один раз на каждом шаге вычисления градиента. Получающаяся процедура обучения описана в алгоритме 18.1. Из-за высокой стоимости приработки марковских цепей во внутреннем цикле эта процедура вычислительно неосуществима, но она может служить отправной точкой, на аппроксимацию которой нацелены более практичные алгоритмы.

Алгоритм 18.1. Наивный MCMC-алгоритм максимизации логарифмического правдоподобия с неразрешимой статистической суммой посредством градиентного восхождения

Установить размер шага ε равным малому положительному числу.
Установить число шагов выборки по Гиббсу k достаточно большим для приработки. Для обучения ОМБ на небольшом фрагменте изображения может хватить значения 100.
while не сошелся **do**
  Выбрать мини-пакет m примеров $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ из обучающего набора
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$.
  Инициализировать набор m примеров $\{\tilde{\mathbf{x}}^{(1)}, ..., \tilde{\mathbf{x}}^{(m)}\}$ случайными значениями (выбранными, например, из равномерного или нормального распределения или, возможно, из распределения, маргиналы которого совпадают с маргиналами модели).
  for $i = 1$ to k **do**
    for $j = 1$ to m **do**

$\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs\_update}(\tilde{\mathbf{x}}^{(j)})$
end for
end for
$\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta})$.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \varepsilon \mathbf{g}$
end while

Мы можем рассматривать подход MCMC к максимальному правдоподобию как попытку уравновесить две силы, одна из которых толкает распределение модели вверх там, где имеются данные, а другая толкает его вниз там, где имеются примеры, выбранные из модели. Этот процесс иллюстрируется на рис. 18.1. Две силы соответствуют максимизации $\log \tilde{p}$ и минимизации $\log Z$. Возможно несколько аппроксимаций отрицательной фазы, каждую из которых можно рассматривать как попытку сделать эту фазу вычислительно более простой, но одновременно и как толкание ее вниз не в тех точках, что нужно.

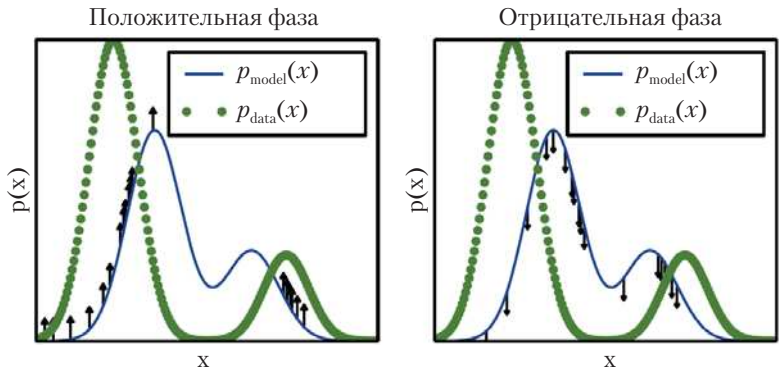

Рис. 18.1 ❖ Взгляд на алгоритм 18.1 с точки зрения положительной и отрицательной фаз. (*Слева*) В положительной фазе мы выбираем точки из распределения данных и толкаем вверх их ненормированную вероятность. Это означает, что точки, которые с вероятностью принадлежат данным, проталкиваются выше вверх. (*Справа*) В отрицательной фазе мы выбираем точки из модельного распределения и толкаем вниз их ненормированную вероятность. Это противодействует стремлению положительной фазы просто всюду прибавить большую постоянную к ненормированной вероятности. Если распределение данных и модельное распределение совпадают, то у положительной фазы такие же шансы поднять точку вверх, как у отрицательной – опустить вниз. Если такое происходит, то градиент математического ожидания обнуляется, и обучение следует остановить

Поскольку в отрицательной фазе примеры выбираются из модельного распределения, то мы можем интерпретировать ее как нахождение точек, в которые модель сильно верит. Так как действие отрицательной фазы сводится к уменьшению вероятности этих точек, обычно они рассматриваются как неверные представления модели о мире. В литературе эти точки часто называют «галлюцинациями», или «воображаемыми частицами» (fantasy particles). На самом деле отрицательная фаза была предложена как

возможное объяснение снов у человека и других животных (Crick and Mitchison, 1983). Идея в том, что мозг хранит вероятностную модель мира и следует в направлении градиента $\log \tilde{p}$, сталкиваясь с реальными событиями в состоянии бодрствования, и в направлении отрицательного градиента $\log \tilde{p}$, стремясь минимизировать $\log Z$, когда спит и сталкивается с событиями, выбранными из текущей модели. Такой взгляд на вещи объясняет терминологию, используемую при описании алгоритмов с положительной и отрицательной фазами, но его правильность не доказана нейробиологическими экспериментами. В моделях машинного обучения обычно необходимо использовать положительную и отрицательную фазы одновременно, а не в раздельные периоды бодрствования и фазы быстрого сна. В разделе 19.5 мы увидим, что другие алгоритмы машинного обучения выбирают примеры из модельного распределения для других целей и что такие алгоритмы также могли бы дать объяснение функции сновидений.

При таком понимании роли положительной и отрицательной фаз обучения мы можем попытаться спроектировать более дешевую альтернативу алгоритма 18.1. Основная часть стоимости наивного алгоритма MCMC – стоимость приработки случайным образом инициализированных марковских цепей на каждом шаге. Естественное решение – инициализировать цепь, воспользовавшись распределением, очень близким к модельному, тогда приработка займет меньше времени.

Алгоритм **сопоставительного расхождения** (contrastive divergence) (CD, или CD-k, чтобы показать, что это алгоритм CD с k шагами выборки по Гиббсу) инициализирует марковскую цепь на каждом шаге примерами, выбранными из распределения данных (Hinton, 2000, 2010). Эта идея представлена в алгоритме 18.2. Получение примеров из распределения данных ничего не стоит, потому что они уже присутствуют в наборе данных. Первоначально распределение данных сильно отличается от модельного, поэтому отрицательная фаза не очень точна. Но, к счастью, положительная фаза все-таки может верно увеличивать вероятность данных в модели. Если дать положительной фазе поработать некоторое время, то модельное распределение окажется ближе к распределению данных, и верность негативной фазы начнет расти.

Алгоритм 18.2. Алгоритм сопоставительного расхождения, в котором в качестве процедуры оптимизации используется градиентное восхождение

Установить размер шага ε равным малому положительному числу.
Установить число шагов выборки по Гиббсу k достаточно большим для того, чтобы выборка по схеме марковской цепи из $p(\mathbf{x}; \boldsymbol{\theta})$ перемешивалась при инициализации из p_{data}. Для обучения ОМБ на небольшом фрагменте изображения можно взять значение от 1 до 20.
while не сошелся **do**
  Выбрать мини-пакет m примеров $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ из обучающего набора
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$.
  for $i = 1$ to m **do**
    $\tilde{\mathbf{x}}(i) \leftarrow \mathbf{x}(i)$
  end for
  for $i = 1$ to k **do**
    for $j = 1$ to m **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs\_update}(\tilde{\mathbf{x}}^{(j)})$
    end for

end for

$$\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta}).$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \varepsilon \mathbf{g}$$

end while

Разумеется, алгоритм CD по-прежнему является лишь приближением к правильной отрицательной фазе. Основная причина, по которой CD качественно не справляется с реализацией отрицательной фазы, заключается в невозможности подавить области высокой вероятности, далекие от реальных обучающих примеров. Такие области, в которых вероятность в модели высокая, а в истинном порождающем данные распределении низкая, называются **паразитными модами**. На рис. 18.2 показано, почему это происходит. Дело в том, что моды модельного распределения, далекие от распределения данных, посещаются марковскими цепями, инициализированными в обучающих точках, только если k очень велико.

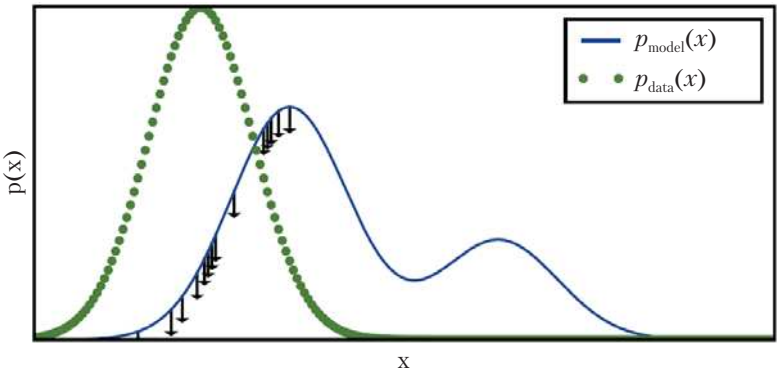

Рис. 18.2 ❖ Паразитная мода. Иллюстрация того, как отрицательная фаза сопоставительного расхождения (алгоритм 18.2) не справляется с подавлением паразитных мод. Паразитной называется мода, присутствующая в модельном распределении, но отсутствующая в истинном распределении данных. Поскольку в алгоритме сопоставительного расхождения марковские цепи инициализируются по точкам из распределения данных и работают всего несколько шагов, то маловероятно, что они посетят моды модели, далеко отстоящие от данных. Это означает, что при выборке из модели мы иногда будем получать примеры, не похожие на данные. Кроме того, из-за расходования части массы вероятности на эти моды модель будет испытывать трудности с размещением областей высокой вероятности в правильных модах. Для наглядности на этом рисунке используется несколько упрощенное понятие расстояния – паразитная мода далеко отстоит от правильной моды вдоль горизонтальной оси в $\mathbb{R}$. Это соответствует марковской цепи, которая производит локальные перемещения с единственной случайной величиной x из $\mathbb{R}$. В большинстве глубоких вероятностных моделей марковские цепи основаны на выборке по Гиббсу и могут нелокально перемещать любую величину, но не все сразу. Для таких задач обычно лучше рассматривать не евклидово, а редакторское расстояние между модами. Однако редакторское расстояние в многомерном пространстве трудно изобразить на двумерном рисунке

В работе Carreira-Perpiñan and Hinton (2005) экспериментально показано, что CD-оценка является смещенной для ограниченных и полностью видимых машин Больцмана в том смысле, что сходится не к тем же точкам, что оценка максимального правдоподобия. Авторы замечают, что поскольку смещение невелико, то алгоритм CD можно было бы использовать как дешевый способ инициализации модели, а затем уточнить модель, применяя более дорогостоящие MCMC-методы. В работе Bengio and Delalleau (2009) показано, что CD можно интерпретировать как отбрасывание наименьших членов правильного градиента MCMC-обновления, объясняющего смещение.

Алгоритм CD полезен для обучения мелких моделей типа ОМБ. Собрав несколько таких моделей, можно инициализировать более глубокие модели, например глубокие сети доверия или глубокие машины Больцмана. Но CD мало чем может помочь в непосредственном обучении более глубоких моделей. Все дело в трудности получения примеров скрытых блоков при наличии примеров видимых блоков. Поскольку скрытые блоки не включаются в данные, инициализация по обучающим примерам не решает проблему. Даже если видимые блоки инициализированы на основе данных, мы все равно должны приработать марковскую цепь, чтобы получить выборку из распределения скрытых блоков при условии видимых примеров.

Можно считать, что алгоритм CD штрафует модель за наличие марковской цепи, которая быстро изменяет вход, если тот поступает из данных. Это означает, что обучение с помощью CD чем-то напоминает обучение автокодировщика. Несмотря на то что смещение CD больше, чем у некоторых других методов обучения, этот алгоритм может быть полезен для предобучения мелких моделей, которые впоследствии собираются в стек. Объясняется это тем, что предшествующие модели в стеке копируют больше информации в свои латентные переменные, делая ее доступной последующим моделям. Это следует рассматривать скорее как часто эксплуатируемый побочный эффект обучения с помощью CD, нежели как принципиальную особенность, заложенную в проект.

В работе Sutskever and Tieleman (2010) показано, что направление обновления в CD не совпадает с направлением градиента какой-либо функции. В результате CD может зациклиться, но на практике это не представляет серьезной проблемы.

Другая стратегия, решающая многие проблемы, присущие CD, – инициализировать марковские цепи на каждом шаге градиентного спуска состояниями с предыдущего шага. Впервые этот подход получил распространение под названием **стохастической максимизации правдоподобия** (СМП) в прикладной математике и статистике (Younes, 1998), а впоследствии был независимо открыт в сообществе глубокого обучения под названием **устойчивое сопоставительное расхождение** (persistent contrastive divergence – PCD или PCD-k, чтобы показать, что используется k шагов выборки по Гиббсу на каждое обновление) (Tieleman, 2008). См. алгоритм 18.3. Основная идея состоит в том, что коль скоро шаги алгоритма стохастического градиентного спуска малы, модели, построенные на двух соседних шагах, будут похожи. Отсюда следует, что примеры, выбранные из распределения предыдущей модели, будут очень близки к настоящим примерам из распределения текущей модели, так что марковская цепь, инициализированная этими примерами, не потребует много времени для приработки.

Поскольку все марковские цепи непрерывно обновляются на протяжении всего процесса обучения, а не перезапускаются на каждом шаге вычисления градиента, то

они могут забрести достаточно далеко, чтобы обнаружить все моды модели. Поэтому СМП оказывается гораздо устойчивее к формированию моделей с паразитными модами, чем CD. Кроме того, благодаря возможности запоминать состояние всех переменных, из которых производится выборка, как видимых, так и латентных, СМП поставляет начальные данные для скрытых и видимых блоков. CD может обеспечить инициализацию только видимых блоков, поэтому в глубоких моделях требуется приработка. СМП способен обучать глубокие модели более эффективно. В работе Marlin et al. (2010) проведено сравнение СМП со многими другими критериями, описанными в этой главе. Показано, что СМП дает наилучшее логарифмическое правдоподобие на тестовом наборе для ОМБ и что если скрытые блоки ОМБ используются в качестве признаков для SVM-классификатора, то СМП дает наилучшую верность классификации.

Алгоритм 18.3. Алгоритм стохастической максимизации правдоподобия (устойчивого сопоставительного расхождения), в котором в качестве процедуры оптимизации используется градиентное восхождение

Установить размер шага ε равным малому положительному числу.

Установить число шагов выборки по Гиббсу k достаточно большим для того, чтобы выборка по схеме марковской цепи из $p(\mathbf{x}; \boldsymbol{\theta} + \varepsilon\mathbf{g})$ приработалась, начав с примеров из $p(\mathbf{x}; \boldsymbol{\theta})$. Для обучения ОМБ на небольшом фрагменте изображения можно взять значение 1, для более сложной модели, например глубокой сети доверия, – от 5 до 50.

Инициализировать набор m примеров $\{\tilde{\mathbf{x}}^{(1)}, ..., \tilde{\mathbf{x}}^{(m)}\}$ случайными значениями (выбранными, например, из равномерного или нормального распределения или, возможно, из распределения, маргиналы которого совпадают с маргиналами модели).

while не сошелся **do**

  Выбрать мини-пакет m примеров $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ из обучающего набора

  $\mathbf{g} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$.

  for $i = 1$ to k **do**

    for $j = 1$ to m **do**

      $\tilde{\mathbf{x}}^{(j)} \leftarrow$ gibbs_update($\tilde{\mathbf{x}}^{(j)}$)

    end for

  end for

  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta})$.

  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \varepsilon\mathbf{g}$

end while

СМП может утратить верность, если стохастический градиентный алгоритм перемещает модель настолько быстро, что марковская цепь не успевает прирабатываться между шагами. Это может случиться, если k слишком мало или ε слишком велико. К сожалению, допустимый диапазон значений сильно зависит от задачи. Неизвестно, как можно формально проверить, успешно ли прирабатывается цепь между шагами. Субъективно, если скорость обучения слишком высока для выбранного числа шагов выборки по Гиббсу, то оператор-человек будет наблюдать, что дисперсия примеров

в отрицательной фазе гораздо больше между шагами вычисления градиента, чем между различными марковскими цепями. Например, модель, обученная на наборе данных MNIST, на одном шаге может выбрать исключительно цифры 7. Тогда процесс обучения сильно опустит вниз моду, соответствующую семерке, и на следующем шаге модель выберет только цифры 9.

При вычислении выборки из модели, обученной с помощью СМП, следует проявлять осторожность. Производить выборку следует, начиная с новой марковской цепи, инициализированной в случайной начальной точке, после того как обучение модели закончено. На примеры, присутствующие в запомненных отрицательных цепях, использованных для обучения, оказали влияние несколько последних версий модели, поэтому может показаться, что емкость модели больше, чем на самом деле.

В работе Berglund and Raiko (2013) поставлены эксперименты для изучения смещения и дисперсии оценок градиента, полученных методами CD и СМП. Показано, что CD дает меньшую дисперсию, чем оценка, основанная на точной выборке. У СМП дисперсия выше. Причина низкой дисперсии CD – в том, что в этом алгоритме одни и те же обучающие примеры используются в положительной, и в отрицательной фазах. Если в отрицательной фазе производить инициализацию на других обучающих данных, то дисперсия будет выше, чем у оценки, основанной на точной выборке.

Все методы, основанные на применении MCMC для выборки из модели, в принципе, можно использовать почти с любым вариантом MCMC. Это означает, что такие алгоритмы, как СМП, можно улучшить, применив любой усовершенствованный MCMC-метод из числа описанных в главе 17, например параллельное темперирование (Desjardins et al., 2010; Choet al., 2010).

Один из подходов к ускорению перемешивания в ходе обучения опирается не на изменение технологии выборки методом Монте-Карло, а на выборе другой параметризации модели и функции стоимости. В алгоритме Fast PCD, или FPCD (Tieleman and Hinton, 2009) параметры $\boldsymbol{\theta}$ традиционной модели заменяются выражением

$$\boldsymbol{\theta} = \boldsymbol{\theta}^{(\text{slow})} + \boldsymbol{\theta}^{(\text{fast})}. \tag{18.16}$$

Теперь параметров вдвое больше, чем раньше, и их поэлементная сумма дает параметры, используемые в исходном определении модели. «Быстрые» параметры $\boldsymbol{\theta}^{(\text{fast})}$ обучаются при гораздо большей скорости обучения, что позволяет им быстро адаптироваться в ответ на отрицательную фазу обучения и подталкивать марковскую цепь к исследованию новой территории. Поэтому цепь быстрее прирабатывается, хотя этот эффект наблюдается только во время обучения, пока быстрые веса могут беспрепятственно изменяться. Как правило, к быстрым весам применяется также тактика снижения весов, поощряющая их сходиться к небольшим значениям, после того как они оставались большими достаточно долго, для того чтобы побудить марковскую цепь к смене моды.

Одно из важных преимуществ MCMC-методов, описанных в этом разделе, состоит в том, что они дают оценку градиента $\log Z$, и потому мы можем представить задачу в виде суммы вкладов $\log \tilde{p}$ и $\log Z$. Затем можно использовать любой другой метод для обработки $\log \tilde{p}(\mathbf{x})$ и просто прибавить к вычисленному им градиенту наш градиент в отрицательной фазе. В частности, это означает, что в положительной фазе можно использовать методы, которые дают только нижнюю границу $\tilde{p}$. Большинство других методов работы с $\log Z$, представленных в этой главе, несовместимо с методами в положительной фазе, основанными на оценке границы.

18.3. Псевдоправдоподобие

Аппроксимации статистической суммы и ее градиента методами Монте-Карло прямо направлены на преодоление связанных со статистической суммой трудностей. В других подходах эта проблема обходится посредством обучения модели без вычисления статистической суммы. Многие такие подходы основаны на возможности легко вычислить отношения вероятностей в неориентированной вероятностной модели. Связано это с тем, что статистическая сумма, входящая в числитель и знаменатель дроби, сокращается:

$$\frac{p(\mathbf{x})}{p(\mathbf{y})} = \frac{\frac{1}{Z}\tilde{p}(\mathbf{x})}{\frac{1}{Z}\tilde{p}(\mathbf{y})} = \frac{\tilde{p}(\mathbf{x})}{\tilde{p}(\mathbf{y})}. \tag{18.17}$$

Псевдоправдоподобие основано на том, что условные вероятности имеют вид такого отношения, поэтому их можно вычислить, не зная статистическую сумму. Предположим, что мы разбили $\mathbf{x}$ на $\mathbf{a}$, $\mathbf{b}$ и $\mathbf{c}$, где $\mathbf{a}$ содержит величины, чье условное распределение мы хотим найти, $\mathbf{b}$ – обусловливающие величины, а $\mathbf{c}$ – величины, не являющиеся частью запроса:

$$p(\mathbf{a} \mid \mathbf{b}) = \frac{p(\mathbf{a}, \mathbf{b})}{p(\mathbf{b})} = \frac{p(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a}, \mathbf{c}} p(\mathbf{a}, \mathbf{b}, \mathbf{c})} = \frac{\tilde{p}(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a}, \mathbf{c}} \tilde{p}(\mathbf{a}, \mathbf{b}, \mathbf{c})}. \tag{18.18}$$

Отсюда требуется исключить $\mathbf{a}$, эта операция может быть выполнена очень эффективно, при условии что $\mathbf{a}$ и $\mathbf{c}$ содержат немного величин. В предельном случае $\mathbf{a}$ состоит всего из одной величины, а $\mathbf{c}$ пусто, так что требуется вычислить $\tilde{p}$ лишь столько раз, сколько значений может принимать одна случайная величина.

К сожалению, чтобы вычислить логарифмическое правдоподобие, нам нужно исключать большие множества величин. Если всего имеется n величин, то требуется исключить множество размера $n - 1$. Согласно цепному правилу вероятностей:

$$\log p(\mathbf{x}) = \log p(x_1) + \log p(x_2 \mid x_1) + \ldots + p(x_n \mid \mathbf{x}_{1:n-1}). \tag{18.19}$$

В данном случае мы взяли наименьшее возможное $\mathbf{a}$, но $\mathbf{c}$ может составлять $\mathbf{x}_{2:n}$. А что, если просто переместить $\mathbf{c}$ в $\mathbf{b}$, чтобы уменьшить вычислительную стоимость? Тогда получится целевая функция **псевдоправдоподобия** (Besag, 1975), основанная на предсказании значения признака x_i при условии всех остальных признаков $\boldsymbol{x}_{-i}$:

$$\sum_{i=1}^{n} \log p(x_i \mid \boldsymbol{x}_{-i}). \tag{18.20}$$

Если каждая случайная величина может принимать k значений, то для вычисления $\tilde{p}$ потребуется произвести только $k \times n$ операций вместо k^n операций, необходимых для вычисления статистической суммы.

На первый взгляд, это кажется беспринципным трюком, но можно доказать, что оценка, полученная максимизацией псевдоправдоподобия, асимптотически состоятельная (Mase, 1995). Конечно, если набор данных нельзя назвать большой выборкой, то поведение псевдоправдоподобия может отличаться от оценки максимального правдоподобия.

Мы можем предпочесть вычислительную сложность отклонению от оценки максимального правдоподобия, воспользовавшись **оценкой обобщенного псевдоправ-**

доподобия (Huang and Ogata, 2002), в которой используется m различных множеств $\mathbb{S}^{(i)}$, $i = 1, ..., m$ индексов величин, встречающихся вместе слева от вертикальной черты в выражении условной вероятности. В предельном случае, когда $m = 1$ и $\mathbb{S}^{(1)} = 1, ..., n$, обобщенное псевдоправдоподобие сводится к логарифмическому правдоподобию. В другом предельном случае, когда $m = n$ и $\mathbb{S}^{(i)} = \{i\}$, обобщенное псевдоправдоподобие сводится к псевдоправдоподобию. Целевая функция обобщенного псевдоправдоподобия имеет вид

$$\sum_{i=1}^{m} \log p(\mathbf{x}_{\mathbb{S}^{(i)}} \mid \mathbf{x}_{-\mathbb{S}^{(i)}}). \tag{18.21}$$

Качество алгоритмов, основанных на псевдоправдоподобии, сильно зависит от способа использования модели. Псевдоправдоподобие плохо работает в задачах, где требуется хорошая модель полного совместного распределения $p(\mathbf{x})$, таких, например, как оценивание плотности или выборка. Оно демонстрирует лучшее качество, чем максимальное правдоподобие, в задачах, где на этапе обучения требуются только условные распределения, например для восполнения небольшого числа отсутствующих значений. Методы на основе обобщенного псевдоправдоподобия особенно эффективны, если данные обладают регулярной структурой, позволяющей проектировать множества индексов $\mathbb{S}$, так чтобы улавливались наиболее важные корреляции, и опускать группы величин, корреляция между которыми пренебрежимо мала. Например, в естественных изображениях пиксели, далеко отстоящие друг от друга в пространстве, слабо коррелированы, поэтому можно применить метод обобщенного псевдоправдоподобия, выбирая в качестве $\mathbb{S}$ небольшое пространственно локализованное окно.

Слабое место оценки псевдоправдоподобия – невозможность использовать ее совместно с другими аппроксимациями, которые дают только нижнюю границу $\tilde{p}(\mathbf{x})$, например вариационным выводом, рассматриваемым в главе 19. Дело в том, что $\tilde{p}$ находится в знаменателе. Нижняя граница знаменателя дает только верхнюю границу выражения в целом, а максимизация верхней границы не дает никакого выигрыша. Это затрудняет применение псевдоправдоподобия к таким моделям, как глубокие машины Больцмана, поскольку вариационные методы – один из преобладающих подходов к приближенному исключению многих слоев скрытых переменных, взаимодействующих друг с другом. Тем не менее псевдоправдоподобие находит применение в глубоком обучении, поскольку его можно использовать для обучения однослойных моделей или глубоких моделей с помощью приближенных методов вывода, не опирающихся на оценку нижней границы.

Для псевдоправдоподобия характерна гораздо более высокая стоимость одного шага вычисления градиента, чем для СМП, поскольку приходится явно вычислять все условные распределения. Но обобщенное псевдоправдоподобие и другие подобные критерии все же могут хорошо работать, если при обработке каждого примера вычисляется только одно случайно выбранное условное распределение (Goodfellow et al., 2013b), так что вычислительная стоимость оказывается сопоставимой с СМП.

Хотя оценка псевдоправдоподобия явно не минимизирует $\log Z$, ее тем не менее можно рассматривать как нечто, похожее на отрицательную фазу. Знаменатели в каждом условном распределении приводят к тому, что алгоритм обучения подавляет вероятность всех состояний, в которых только одна переменная отличается от обучающего примера.

Теоретический анализ асимптотической эффективности псевдоправдоподобия см. в работе Marlin and de Freitas (2011).

18.4. Сопоставление рейтингов и сопоставление отношений

Сопоставление рейтингов (score matching) (Hyvärinen, 2005) – еще один способ обучить модель, не оценивая ни статистическую сумму Z, ни ее производные. Происхождение названия связано с терминологией, согласно которой производные логарифма плотности по ее аргументу, $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$, называются **рейтингом** (score). В методе сопоставления рейтингов стратегия заключается в минимизации ожидаемого квадрата разности между градиентом логарифма модельной плотности по входу и градиентом логарифма плотности данных по входу:

$$L(\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{2} \left\| \nabla_{\boldsymbol{x}} \log p_{\text{model}}(\boldsymbol{x}, \boldsymbol{\theta}), -\nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x}) \right\|_2^2, \tag{18.22}$$

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{p_{\text{data}}(\boldsymbol{x})} L(\boldsymbol{x}, \boldsymbol{\theta}), \tag{18.23}$$

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{18.24}$$

При такой целевой функции мы избегаем трудностей дифференцирования статистической суммы Z, поскольку Z не зависит от $\boldsymbol{x}$ и, следовательно, $\nabla_{\mathbf{x}} Z = 0$. Поначалу кажется, что у сопоставления рейтингов другая сложность: чтобы вычислить рейтинг распределения данных, необходимо знать истинное порождающее распределение обучающих данных, p_{data}. По счастью, минимизация ожидаемого значения $L(\boldsymbol{x}; \boldsymbol{\theta})$ эквивалентна минимизации ожидаемого значения выражения

$$\tilde{L}(\boldsymbol{x}, \boldsymbol{\theta}) = \sum_{j=1}^{n} \left(\frac{\partial^2}{\partial x_j^2} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) + \frac{1}{2} \left(\frac{\partial}{\partial x_j} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) \right)^2 \right), \tag{18.25}$$

где n – размерность $\boldsymbol{x}$.

Поскольку в методе сопоставления рейтингов требуется вычислять производные по $\mathbf{x}$, он не применим к моделям с дискретными данными, но латентные переменные модели могут быть дискретными.

Как и псевдоправдоподобие, метод сопоставления рейтингов работает только тогда, когда мы можем непосредственно вычислить функцию $\log \tilde{p}(\mathbf{x})$ и ее производные. Он не совместим с методами, которые дают только нижнюю границу $\log \tilde{p}(\mathbf{x})$, потому что для сопоставления рейтингов нужны производные и вторые производные $\log \tilde{p}(\mathbf{x})$, а нижняя граница не несет никакой информации о производных. Это означает, что метод сопоставления рейтингов неприменим к оцениванию моделей со сложными взаимодействиями между скрытыми блоками, например моделям разреженного кодирования или глубоким машинам Больцмана. И хотя сопоставление рейтингов можно применять для предобучения первого скрытого слоя большой модели, этот метод никогда не применялся в качестве стратегии предобучения более глубоких слоев. Возможно, дело в том, что скрытые слои таких моделей обычно содержат какие-то дискретные переменные.

В методе сопоставления рейтингов нет явной отрицательной фазы, но его можно рассматривать как вариант состязательного расхождения, если использовать марковскую цепь специального вида (Hyvärinen, 2007a). В данном случае марковская цепь определяется не выборкой по Гиббсу, а совсем другим способом, в котором локальные перемещения направляются градиентом. Сопоставление рейтингов эквивалентно алгоритму CD с марковской цепью такого вида, когда величина локальных перемещений стремится к нулю.

В работе Lyu (2009) метод сопоставления рейтингов обобщен на дискретный случай (но в доказательстве была допущена ошибка, исправленная в работе Marlin et al. [2010]). В работе Marlin et al. (2010) обнаружено, что **обобщенное сопоставление рейтингов** (generalized score matching – GSM) не работает в многомерных дискретных пространствах, где наблюдаемая вероятность многих событий равна 0.

Более плодотворный подход к обобщению основных идей сопоставления рейтингов на дискретные данные – **сопоставление отношений** (ratio matching) (Hyvärinen, 2007b). Этот метод применим только к бинарным данным и заключается в минимизации усреднения по всем примерам следующей целевой функции:

$$L^{(\mathrm{RM})}(\boldsymbol{x}, \boldsymbol{\theta}) = \sum_{j=1}^{n} \left(\frac{1}{1 + \dfrac{p_{\mathrm{model}}(\boldsymbol{x}; \boldsymbol{\theta})}{p_{\mathrm{model}}(f(\boldsymbol{x}), j); \boldsymbol{\theta})}} \right)^2, \tag{18.26}$$

где $f(\boldsymbol{x}, j)$ возвращает вектор $\mathbf{x}$, в котором бит в позиции j изменен на противоположный. Чтобы избежать вычисления статистической суммы, в методе сопоставления отношений применен тот же прием, что и в оценке псевдоправдоподобия: в отношении двух вероятностей статистическая сумма сокращается. В работе Marlin et al. (2010) показано, что метод сопоставления отношений превосходит СМП, псевдоправдоподобие и GSM с точки зрения способности обученной модели очищать изображения из тестового набора от шума.

Как и в оценке псевдоправдоподобия, в методе сопоставления отношений требуется n раз вычислить $\tilde{p}$ на каждый пример, поэтому его вычислительная стоимость в расчете на одно обновление примерно в n раз выше, чем стоимость СМП.

Как и оценку псевдоправдоподобия, сопоставление отношений можно интерпретировать как толкание вниз для всех воображаемых состояний, в которых только одна переменная отличается от обучающего примера. Поскольку метод сопоставления отношений применяется исключительно к бинарным данным, это означает, что он воздействует на все воображаемые состояния, находящиеся от обучающих данных на расстоянии Хэмминга 1.

Сопоставление отношений может быть полезно в качестве основы для работы с разреженными данными большой размерности, например векторами счетчиков слов. Для МСМС-методов такие данные составляют проблему, потому что представить данные в плотном формате обходится очень дорого, а компонент выборки МСМС не выдает разреженных данных, пока модель не обучится представлять разреженность в распределении данных. В работе Dauphin and Bengio (2013) эта трудность преодолена путем проектирования несмещенной стохастической аппроксимации сопоставления отношений. Аппроксимация вычисляет только случайно выбранное подмножество членов целевой функции и не требует, чтобы модель порождала полные воображаемые выборки.

Теоретический анализ асимптотической эффективности сопоставления отношений см. в работе Marlin and de Freitas (2011).

18.5. Шумоподавляющее сопоставление рейтингов

Иногда бывает полезно регуляризировать сопоставление рейтингов, аппроксимируя распределение

$$p_{\text{smoothed}}(\boldsymbol{x}) = \int p_{\text{data}}(\boldsymbol{y}) q(\boldsymbol{x} \mid \boldsymbol{y}) d\boldsymbol{y}, \tag{18.27}$$

а не истинное распределение p_{data}. Распределение $q(\boldsymbol{x} \mid \boldsymbol{y})$ описывает искажающий процесс, обычно он формирует $\boldsymbol{x}$, прибавляя небольшой шум к $\boldsymbol{y}$.

Шумоподавляющее сопоставление рейтингов особенно полезно, потому что на практике мы обычно не имеем доступа к истинному p_{data}, а только к эмпирическому распределению, определенному выборкой из p_{data}. При достаточной емкости любая состоятельная оценка превратит p_{model} во множество распределений Дирака с центрами в обучающих точках. Сглаживание с помощью q помогает устранить эту проблему ценой утраты свойства асимптотической состоятельности, описанного в разделе 5.4.5. В работе Kingma and LeCun (2010) описана процедура выполнения регуляризированного сопоставления рейтингов, когда сглаживающее распределение q – нормально распределенный шум.

Напомним (см. раздел 14.5.1), что некоторые алгоритмы обучения автокодировщиков эквивалентны сопоставлению рейтингов или шумоподавляющему сопоставлению рейтингов. Следовательно, эти алгоритмы также решают проблему статистической суммы.

18.6. Шумосопоставительное оценивание

Большинство методов оценивания моделей с вычислительно неразрешимыми статистическими суммами не дает оценки этой суммы. Алгоритмы СМП и CD оценивают только градиент логарифма статистической суммы, а не ее саму. Методы сопоставления рейтингов и псевдоправдоподобия вообще избегают вычисления величин, связанных со статистической суммой.

Шумосопоставительное оценивание (noise-contrastive estimation – NCE) (Gutmann and Hyvarinen, 2010) предлагает другую стратегию. В этом методе распределение вероятности, оцениваемое моделью, представляется явно в виде

$$\log p_{\text{model}}(\mathbf{x}) = \log \tilde{p}_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}) + c, \tag{18.28}$$

где c вводится как аппроксимация $-\log Z(\boldsymbol{\theta})$. Вместо того чтобы оценивать только $\boldsymbol{\theta}$, процедура шумосопоставительного оценивания рассматривает c как еще один параметр и оценивает $\boldsymbol{\theta}$ и c одновременно, применяя один и тот же алгоритм. Результирующее $\log p_{\text{model}}(\mathbf{x})$ может не соответствовать точно корректному распределению вероятности, но приближается к нему по мере улучшения оценки c[1].

[1] Метод NCE применим также к задачам с вычислимой статистической суммой, когда нет нужды вводить дополнительный параметр c. Но наибольший интерес он вызывает как средство оценивания моделей с трудно вычислимыми статистическими суммами.

Такой подход был бы невозможен, если бы в качестве критерия оценки использовалось максимальное правдоподобие. Критерий максимального правдоподобия стремился бы присвоить c как можно большее значение, а не достичь корректного распределения вероятности.

В основу работы NCE положено сведение зад,ачи обучения без учителя, заключающейся в оценивании $p(\mathbf{x})$, к обучению вероятностного бинарного классификатора, в котором одна из категорий соответствует данным, порожденным моделью. Эта задача обучения с учителем конструируется так, что оценка максимального правдоподобия определяет асимптотически состоятельную оценку для исходной задачи.

Точнее говоря, мы вводим второе **распределение шумов** $p_{\text{noise}}(\mathbf{x})$. Оно должно быть легко вычислимо и выборка из него не должна вызывать затруднений. Теперь мы можем построить модель, включающую $\mathbf{x}$ и новую бинарную переменную класса y. Для новой совместной модели мы постулируем:

$$p_{\text{joint}}(y = 1) = {}^1\!/_2, \tag{18.29}$$

$$p_{\text{joint}}(\mathbf{x} \mid y = 1) = p_{\text{model}}(\mathbf{x}), \tag{18.30}$$

и

$$p_{\text{joint}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x}). \tag{18.31}$$

Иными словами, переменная y – переключатель, определяющий, из какого распределения выбирать $\mathbf{x}$: из модели или из шума.

Можно построить аналогичную совместную модель обучающих данных. В этом случае переключатель определяет, выбирается $\mathbf{x}$ из распределения **данных** или из шума. Формально $p_{\text{train}}(y = 1) = {}^1\!/_2$, $p_{\text{train}}(\mathbf{x} \mid y = 1) = p_{\text{data}}(\mathbf{x})$, $p_{\text{train}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x})$.

Теперь можно применить стандартное обучение методом максимального правдоподобия к задаче обучения с учителем, заключающейся в подгонке p_{joint} к p_{train}.

$$\boldsymbol{\theta}, c = \arg\max_{\boldsymbol{\theta}, c} \mathbb{E}_{\mathbf{x}, \mathrm{y} \sim p_{\text{train}}} \log p_{\text{joint}}(y \mid \mathbf{x}). \tag{18.32}$$

Распределение p_{joint} – это, по существу, модель логистической регрессии, примененная к разности логарифмов распределения вероятности модели и шума.

$$p_{\text{joint}}(y = 1 \mid \mathbf{x}) = \frac{p_{\text{model}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x}) + p_{\text{noise}}(\mathbf{x})} \tag{18.33}$$

$$= \frac{1}{1 + \dfrac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}} \tag{18.34}$$

$$= \frac{1}{1 + \exp\left(\log \dfrac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})} \right)} \tag{18.35}$$

$$= \sigma\left(-\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})} \right) \tag{18.36}$$

$$= \sigma(\log p_{\text{model}}(\mathbf{x}) - \log p_{\text{noise}}(\mathbf{x})). \tag{18.37}$$

Таким образом, метод NCE применить просто при условии, что $\log \tilde{p}_{\text{model}}$ легко поддается обратному распространению, и, как было сказано выше, p_{noise} легко вычисляется (для вычисления p_{joint}) и допускает простую выборку (для генерации обучающих данных).

Особенный успех методу NCE сопутствует в задачах с небольшим числом случайных величин, но он неплохо работает и тогда, когда эти величины принимают много разных значений. Например, он успешно применялся к моделированию условного распределения слова при известном контексте (Mnih and Kavukcuoglu, 2013). Слово в данном случае всего одно, хотя и может выбираться из большого словаря.

Применение NCE к задачам с большим числом случайных величин менее эффективно. Классификатор на основе логистической регрессии может отклонить зашумленный пример, выявив всего одну переменную с маловероятным значением. Это означает, что обучение сильно замедляется, после того как p_{model} обучилось базовой маргинальной статистике. Представьте себе обучение модели изображений лиц, в которой в качестве p_{noise} используется неструктурированный гауссов шум. Если p_{model} обучилась распознавать глаза, то она может отклонять почти все зашумленные примеры, так ничего и не узнав о других признаках лица, например ртах.

Ограничение на простоту вычисления и выборки из p_{noise} может оказаться чрезмерно строгим. Если распределение p_{noise} простое, то большинство примеров, скорее всего, будет слишком очевидно отличаться от данных, так что заметного улучшения p_{model} достичь не удастся.

Подобно сопоставлению рейтингов и псевдоправдоподобию, NCE не работает, если известна только нижняя граница $\tilde{p}$. Нижнюю границу можно было бы использовать для получения нижней границы $p_{\text{joint}}(y = 1 \mid \mathbf{x})$, но она позволяет получить лишь верхнюю границу распределения $p_{\text{joint}}(y = 0 \mid \mathbf{x})$, которое встречается в половине членов целевой функции NCE. Точно так же бесполезна нижняя граница p_{noise}, потому что она дает только верхнюю границу $p_{\text{joint}}(y = 1 \mid \mathbf{x})$.

Когда модельное распределение копируется для определения нового распределения шумов перед каждым шагом градиентного спуска, NCE определяет процедуру **самосопоставительного оценивания** (self-contrastive estimation), для которой ожидаемый градиент эквивалентен ожидаемому градиенту максимального правдоподобия (Goodfellow, 2014). Частный случай NCE, когда зашумленные примеры порождаются моделью, наводит на мысль, что оценку максимального правдоподобия можно интерпретировать как процедуру, которая заставляет модель постоянно обучаться отличать реальность от собственных эволюционирующих представлений, тогда как шумосопоставительная оценка достигает некоторого снижения вычислительной стоимости, заставляя модель отличать реальность только от фиксированного эталона (модели шума).

Применение задачи обучения учителем, заключающейся в различении обучающих и порожденных примеров (когда в определении классификатора участвует функция энергии модели), для предоставления градиента модели ранее уже встречалось в разных формах (Welling et al., 2003b; Bengio, 2009).

В основе шумосопоставительного оценивания лежит идея о том, что хорошая порождающая модель должна быть способна отличить данные от шума. С этим тесно связана другая идея: хорошая порождающая модель должна уметь порождать примеры, которые ни один классификатор не сможет отличить от данных. Эта идея ведет к порождающим состязательным сетям (раздел 20.10.4).

18.7. Оценивание статистической суммы

Большая часть этой главы посвящена описанию методов, позволяющих избежать вычисления неразрешимой статистической суммы $Z(\boldsymbol{\theta})$, ассоциированной с неориентированной графической моделью, но в этом разделе мы обсудим несколько методов прямого оценивания этой суммы.

Умение оценивать статистическую сумму нужно в случае, когда мы хотим вычислить нормированное правдоподобие данных. Это важно при *оценивании* модели, мониторинге качества обучения и сравнении моделей между собой.

Пусть, например, имеются две модели: $\mathcal{M}_A$, определяющая распределение вероятности $p_A(\mathbf{x}; \boldsymbol{\theta}_A) = (1/Z_A)\tilde{p}_A(\mathbf{x}; \boldsymbol{\theta}_A)$, и $\mathcal{M}_B$, определяющая распределение вероятности $p_B(\mathbf{x}; \boldsymbol{\theta}_B) = (1/Z_B)\tilde{p}_B(\mathbf{x}; \boldsymbol{\theta}_B)$. Обычный способ сравнить две модели заключается в том, чтобы вычислить и сравнить правдоподобие, которое они назначают тестовому набору независимых и одинаково распределенных данных. Предположим, что тестовый набор состоит из m примеров $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$. Если $\prod_i p_A(\mathrm{x}^{(i)}; \boldsymbol{\theta}_A) > \prod_i p_B(\mathrm{x}^{(i)}; \boldsymbol{\theta}_B)$ или, что то же самое,

$$\sum_i \log p_A(\mathrm{x}^{(i)}; \boldsymbol{\theta}_A) - \sum_i \log p_B(\mathrm{x}^{(i)}; \boldsymbol{\theta}_B) > 0, \tag{18.38}$$

то мы говорим, что $\mathcal{M}_A$ лучше $\mathcal{M}_B$ (или, по крайней мере, что $\mathcal{M}_A$ лучше моделирует данный тестовый набор) в том смысле, что логарифмическое правдоподобие на тестовом наборе выше. К сожалению, для проверки этого условия нужно знать статистическую сумму. Для вычисления выражения (18.38) требуется знать логарифм вероятности, которую модель назначает каждой точке, а для этого, в свою очередь, нужно вычислить статистическую сумму. Ситуацию можно немного упростить, переписав (18.38) в виде, где нужно знать только **отношение** статистических сумм двух моделей:

$$\sum_i \log p_A(\mathbf{x}^{(i)}; \boldsymbol{\theta}_A) - \sum_i \log p_B(\mathbf{x}^{(i)}; \boldsymbol{\theta}_B) = \sum_i \left(\log \frac{\tilde{p}_A(\mathbf{x}^{(i)}; \boldsymbol{\theta}_A)}{\tilde{p}_B(\mathbf{x}^{(i)}; \boldsymbol{\theta}_B)} \right) - m \log \frac{Z(\boldsymbol{\theta}_A)}{Z(\boldsymbol{\theta}_B)}. \tag{18.39}$$

Таким образом, мы можем установить, что модель $\mathcal{M}_A$ лучше $\mathcal{M}_B$, не зная статистическую сумму каждой модели, а зная только их отношение. Как мы вскоре увидим, это отношение можно оценить с помощью выборки по значимости, при условии что обе модели похожи.

Однако если бы мы захотели вычислить фактическую вероятность тестовых данных в модели $\mathcal{M}_A$ или $\mathcal{M}_B$, то пришлось бы вычислять значения самых статистических сумм. Но если бы мы знали отношение статистических сумм $r = Z(\boldsymbol{\theta}_B)/Z(\boldsymbol{\theta}_A)$ и фактическое значение хотя бы одной из них, скажем $Z(\boldsymbol{\theta}_A)$, то могли бы вычислить значение другой:

$$Z(\boldsymbol{\theta}_B) = rZ(\boldsymbol{\theta}_A) = \frac{Z(\boldsymbol{\theta}_B)}{Z(\boldsymbol{\theta}_A)} Z(\boldsymbol{\theta}_A). \tag{18.40}$$

Для оценки статистической суммы можно воспользоваться методом Монте-Карло, например простой выборкой по значимости. Мы опишем этот подход для непрерывных случайных величин, но он легко переносится и на дискретные величины – нужно только заменить интегрирование суммированием. Воспользуемся вспомогательным распределением $p_0(\mathbf{x}) = (1/Z_0)\tilde{p}_0(\mathbf{x})$, для которого возможно произвести выборку и вычислить как статистическую сумму Z_0, так и ненормированное распределение $\tilde{p}_0(\mathbf{x})$.

$$Z_1 = \int \tilde{p}_1(\mathbf{x}) d\mathbf{x} \tag{18.41}$$

$$= \int \frac{p_0(\mathbf{x})}{p_0(\mathbf{x})} \tilde{p}_1(\mathbf{x}) d\mathbf{x} \tag{18.42}$$

$$= Z_0 \int p_0(\mathbf{x}) \frac{\tilde{p}_1(\mathbf{x})}{\tilde{p}_0(\mathbf{x})} d\mathbf{x} \tag{18.43}$$

$$\hat{Z}_1 = \frac{Z_0}{K} \sum_{k=1}^{K} \frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})}, \text{ где } \mathbf{x}^{(k)} \sim p_0. \tag{18.44}$$

В последней строчке записана оценка Монте-Карло $\hat{Z}_1$ интеграла, полученная посредством выборки из $p_0(\mathbf{x})$ с последующим умножением каждого примера на вес, равный отношению ненормированного распределения $\tilde{p}_1$ к вспомогательному p_0.

Этот подход позволяет также оценить отношение статистических сумм в виде:

$$\frac{1}{K} \sum_{k=1}^{K} \frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})}, \text{ где } \mathbf{x}^{(k)} \sim p_0. \tag{18.45}$$

Затем это значение можно использовать непосредственно для сравнения двух моделей, как описано в формуле (18.39).

Если распределение p_0 близко к p_1, то выражение (18.44) может дать эффективный способ оценивания статистической суммы (Minka, 2005). К сожалению, в большинстве случаев p_1 не только сложное (обычно многомодальное), но и определено в пространстве высокой размерности. Трудно найти распределение p_0, которое было бы, с одной стороны, достаточно простым для вычисления, а с другой – достаточно близким к p_1, чтобы дать высококачественную аппроксимацию. Если p_0 и p_1 не близки, то у большинства примеров из p_0 будет низкая вероятность относительно p_1, поэтому они вносят (сравнительно) пренебрежимо малый вклад в сумму в выражении (18.44).

Присутствие в этой сумме небольшого числа примеров со значительными весами приводит к оценке низкого качества из-за высокой дисперсии. Количественно это можно понять, оценив дисперсию нашей оценки $\hat{Z}_1$:

$$\hat{\mathrm{Var}}(\hat{Z}_1) = \frac{Z_0}{K^2} \sum_{k=1}^{K} \left(\frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} - \hat{Z}_1 \right)^2. \tag{18.46}$$

Эта величина будет наибольшей, когда имеется значительное расхождение между значениями весов $\tilde{p}_1(\mathbf{x}^{(k)})/\tilde{p}_0(\mathbf{x}^{(k)})$.

Теперь обратимся к двум родственным стратегиям, предназначенным для решения трудной задачи оценивания статистической суммы сложных распределений в многомерных пространствах: выборке по значимости с отжигом и мостиковой выборке. Обе начинаются с простой выборки по значимости, описанной выше, и стремятся решить проблему непохожести вспомогательного распределения p_0 на p_1, вводя промежуточные распределения, наводящие мост между p_0 и p_1.

18.7.1. Выборка по значимости с отжигом

В ситуациях, когда расстояние Кульбака–Лейблера $D_{\mathrm{KL}}(p_0 \| p_1)$ велико (т. е. перекрытие между p_0 и p_1 мало), стратегия **выборки по значимости с отжигом** (annealed importance sampling – AIS) пытается перебросить мост между двумя распределения-

ми, вводя промежуточные распределения (Jarzynski, 1997; Neal, 2001). Рассмотрим последовательность распределений p_{η_0}, ..., p_{η_n}, где $0 = \eta_0 < \eta_1 < ... < \eta_{n-1} < \eta_n = 1$, такую, что первое распределение в последовательности совпадает с p_0, а последнее – с p_1.

Это позволяет оценить статистическую сумму многомодального распределения в многомерном пространстве (например, распределения, соответствующего обученной ОМБ). Мы начинаем с более простой модели с известной статистической суммой (например, ОМБ с нулевыми весами) и оцениваем отношение между статистическими суммами двух моделей. Оценка этого отношения основана на оценке отношений последовательности многих похожих распределений, например последовательности ОМБ с весами, образующими интерполяцию между нулевыми и обученными весами.

Отношение Z_1/Z_0 можно записать в виде:

$$\frac{Z_1}{Z_0} = \frac{Z_1}{Z_0}\frac{Z_{\eta_1}}{Z_{\eta_1}}\cdots\frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-1}}} \tag{18.47}$$

$$= \frac{Z_{\eta_1}}{Z_0}\frac{Z_{\eta_2}}{Z_{\eta_1}}\cdots\frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-2}}}\frac{Z_1}{Z_{\eta_{n-1}}} \tag{18.48}$$

$$= \prod_{j=0}^{n-1}\frac{Z_{\eta_{j+1}}}{Z_{\eta_j}}. \tag{18.49}$$

При условии что распределения p_{η_j} и $p_{\eta_{j+1}}$ достаточно близки для всех $0 \le j \le n-1$, мы можем надежно оценить каждый сомножитель $Z_{\eta_{j+1}}/Z_{\eta_j}$ с помощью простой выборки по значимости, а затем воспользоваться этими оценками для получения оценки Z_1/Z_0.

Откуда берутся эти промежуточные распределения? Так же как и вспомогательное распределение p, они задаются проектировщиком, т. е. специально строятся под конкретную задачу. Часто в качестве промежуточных распределений берут взвешенное среднее геометрическое целевого распределения p_1 и начального вспомогательного распределения p_0, для которого статистическая сумма известна:

$$p_{\eta_j} \propto p_1^{\eta_j} p_0^{1-\eta_j}. \tag{18.50}$$

Для выборки из этих промежуточных распределений мы определяем последовательность переходных функций марковской цепи $T_{\eta_j}(\boldsymbol{x}' \mid \boldsymbol{x})$, которые задают условное распределение вероятности перехода в $\boldsymbol{x}'$ при условии нахождения в $\boldsymbol{x}$. Оператор перехода $T_{\eta_j}(\boldsymbol{x}' \mid \boldsymbol{x})$ определяется так, чтобы $p_{\eta_j}(\boldsymbol{x})$ было инвариантным:

$$p_{\eta_j}(\boldsymbol{x}) = \int p_{\eta_j}(\boldsymbol{x}')\, T_{\eta_j}(\boldsymbol{x} \mid \boldsymbol{x}')d\boldsymbol{x}'. \tag{18.51}$$

Эти переходы можно строить в виде любого MCMC-метода (например, Метрополиса–Гастингса или Гиббса), в т. ч. и включающего несколько проходов по всем случайным величинам или какую-то другую форму итерации.

Стратегия выборки AIS заключается в том, чтобы произвести выборку из p_0, а затем с помощью операторов перехода последовательно порождать выборки из промежуточных распределений, пока не дойдем до выборки из целевого распределения p_1:

❍ for $k = 1 ... K$
  – произвести выборку $\boldsymbol{x}_{\eta_1}^{(k)} \sim p_0(\mathbf{x})$
  – произвести выборку $\boldsymbol{x}_{\eta_2}^{(k)} \sim T_{\eta_1}(\boldsymbol{x}_{\eta 2}^{(k)} \mid \boldsymbol{x}_{\eta 1}^{(k)})$
  – ...

– произвести выборку $\boldsymbol{x}_{\eta_{n-1}}^{(k)} \sim T_{\eta_{n-2}}(\boldsymbol{x}_{\eta_{n-1}}^{(k)} \mid \boldsymbol{x}_{\eta_{n-2}}^{(k)})$

– произвести выборку $\boldsymbol{x}_{\eta_n}^{(k)} \sim T_{\eta_{n-1}}(\boldsymbol{x}_{\eta_n}^{(k)} \mid \boldsymbol{x}_{\eta_{n-1}}^{(k)})$

❍ end

Для k-ой выборки мы можем вывести вес значимости, перемножив веса значимости для переходов между промежуточными распределениями в выражении (18.49):

$$w^{(k)} = \frac{\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\boldsymbol{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_2}(\boldsymbol{x}_{\eta_2}^{(k)})}{\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_2}^{(k)})} \cdots \frac{\tilde{p}_1(\boldsymbol{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_n}^{(k)})}. \tag{18.52}$$

Во избежание численных трудностей, в т. ч. переполнения, лучше вычислять $\log w^{(k)}$, складывая и вычитая логарифмы вероятностей, а не сам вес $w^{(k)}$, для чего требуется выполнять умножение и деление.

При так определенной процедуре выборки и вычисления весов значимости получается следующая оценка отношения статистических сумм:

$$\frac{Z_1}{Z_0} \approx \frac{1}{K} \sum_{k=1}^{K} w^{(k)}. \tag{18.53}$$

Для проверки того, что эта процедура определяет корректную схему выборки по значимости, можно показать (Neal, 2001), что она соответствует простой выборке по значимости в расширенном пространстве состояний, когда точки выбираются из произведения пространств $[\boldsymbol{x}_{\eta_1}, ..., \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1]$. Для этого определим распределение в расширенном пространстве:

$$\tilde{p}(\boldsymbol{x}_{\eta_1}, ..., \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1) \tag{18.54}$$

$$= \tilde{p}_1(\boldsymbol{x}_1)\tilde{T}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_{n-1}} \mid \boldsymbol{x}_1)\tilde{T}_{\eta_{n-2}}(\boldsymbol{x}_{\eta_{n-2}} \mid \boldsymbol{x}_{\eta_{n-1}}) \ldots \tilde{T}_{\eta_1}(\boldsymbol{x}_{\eta_1} \mid \boldsymbol{x}_{\eta_2}), \tag{18.55}$$

где $\tilde{T}_a$ – обращение оператора перехода T_a (путем применения правила Байеса):

$$\tilde{T}_a(\boldsymbol{x}' \mid \boldsymbol{x}) = \frac{p_a(\boldsymbol{x}')}{p_a(\boldsymbol{x})} T_a(\boldsymbol{x} \mid \boldsymbol{x}') = \frac{\tilde{p}_a(\boldsymbol{x}')}{\tilde{p}_a(\boldsymbol{x})} T_a(\boldsymbol{x} \mid \boldsymbol{x}'). \tag{18.56}$$

Подставляя это в выражение для совместного распределения в расширенном пространстве состояний (18.55), имеем:

$$\tilde{p}(\boldsymbol{x}_{\eta_1}, ..., \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1) \tag{18.57}$$

$$= \tilde{p}_1(\boldsymbol{x}_1) \frac{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_{n-1}})}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_1)} T_{\eta_{n-1}}(\boldsymbol{x}_1 \mid \boldsymbol{x}_{\eta_{n-1}}) \prod_{i=1}^{n-2} \frac{\tilde{p}_{\eta_i}(\boldsymbol{x}_{\eta_i})}{\tilde{p}_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}})} T_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}} \mid \boldsymbol{x}_{\eta_i}) \tag{18.58}$$

$$= \frac{\tilde{p}_1(\boldsymbol{x}_1)}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_1)} T_{\eta_{n-1}}(\boldsymbol{x}_1 \mid \boldsymbol{x}_{\eta_{n-1}}) \tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_1}) \prod_{i=1}^{n-2} \frac{\tilde{p}_{\eta_{i+1}}(\boldsymbol{x}_{\eta_{i+1}})}{\tilde{p}_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}})} T_{\eta_i}(\boldsymbol{x}_{\eta_{i+1}} \mid \boldsymbol{x}_{\eta_i}). \tag{18.59}$$

Теперь мы получили способ произвести выборку из совместного вспомогательного распределения q в расширенном пространстве, которое определяется формулой:

$$q(\boldsymbol{x}_{\eta_1}, ..., \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1) = p_0(\boldsymbol{x}_{\eta_1}) T_{\eta_1}(\boldsymbol{x}_{\eta_2} \mid \boldsymbol{x}_{\eta_1}) \ldots T_{\eta_{n-1}}(\boldsymbol{x}_1 \mid \boldsymbol{x}_{\eta_{n-1}}). \tag{18.60}$$

Мы имеем совместное распределение в расширенном пространстве, описываемое формулой (18.59). Если взять $q(\boldsymbol{x}_{\eta_1}, ..., \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1)$ в качестве вспомогательного распределения в расширенном пространстве, из которого мы можем произвести выборку, то останется определить веса значимости:

$$w^{(k)} = \frac{\tilde{p}(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1)}{q(\boldsymbol{x}_{\eta_1}, \ldots, \boldsymbol{x}_{\eta_{n-1}}, \boldsymbol{x}_1)} = \frac{\tilde{p}_1(\boldsymbol{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\boldsymbol{x}_{\eta_{n-1}}^{(k)})} \cdots \frac{\tilde{p}_{\eta_2}(\boldsymbol{x}_{\eta_2}^{(k)})}{\tilde{p}_1(\boldsymbol{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_1}(\boldsymbol{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\boldsymbol{x}_0^{(k)})}. \tag{18.61}$$

Это те же веса, что предложенные в алгоритме AIS. Следовательно, AIS можно интерпретировать как простую выборку по значимости в применении к расширенному состоянию, и его корректность немедленно вытекает из корректности выборки по значимости.

Выборка по значимости с отжигом впервые была предложена в работе Jarzynski (1997), а затем независимо в работе Neal (2001). В настоящее время это самый распространенный способ оценивания статистической суммы для неориентированных вероятностных моделей. Возможно, это больше связано с авторитетом авторов публикации Salakhutdinov and Murray (2008), в которой описывается его применение к оцениванию статистической суммы ограниченных машин Больцмана и глубоких сетей доверия, чем с внутренне присущими методу достоинствами.

Обсуждение свойств оценки AIS (в частности, дисперсии и эффективности) имеется в работе Neal (2001).

18.7.2. Мостиковая выборка

Мостиковая выборка (bridge sampling) (Bennett, 1976) – еще один метод, который, как и AIS, направлен на преодоление недостатков выборки по значимости. Но вместо цепочки промежуточных распределений в методе мостиковой выборки используется единственное распределение p_*, называемое мостиком, для интерполяции между распределением с известной статистической суммой p_0 и распределением p_1, для которого мы пытаемся оценить статистическую сумму Z_1.

В методе мостиковой выборки отношение Z_1/Z_0 оценивается как отношение ожидаемых весов значимости между $\tilde{p}_0$ и $\tilde{p}_*$ и между $\tilde{p}_1$ и $\tilde{p}_*$:

$$\frac{Z_1}{Z_0} \approx \sum_{k=1}^{K} \frac{\tilde{p}_*(\boldsymbol{x}_0^{(k)})}{\tilde{p}_0(\boldsymbol{x}_0^{(k)})} \Bigg/ \sum_{k=1}^{K} \frac{\tilde{p}_*(\boldsymbol{x}_1^{(k)})}{\tilde{p}_1(\boldsymbol{x}_1^{(k)})}. \tag{18.62}$$

Если мостиковое распределение p_* тщательно выбрано таким образом, что оно сильно перекрывается и с p_0, и с p_1, то мостиковая выборка допускает гораздо большее расстояние между двумя распределениями (точнее, $D_{\mathrm{KL}}(p_0 \| p_1)$), чем при стандартной выборке по значимости.

Можно показать, что оптимальное мостиковое распределение описывается формулой $p_*^{(opt)}(\mathbf{x}) \propto \big(\tilde{p}_0(\boldsymbol{x})\tilde{p}_1(\boldsymbol{x})\big)/\big(r\tilde{p}_0(\boldsymbol{x}) + \tilde{p}_1(\boldsymbol{x})\big)$, где $r = Z_1/Z_0$. На первый взгляд кажется, что это бесполезная конструкция, потому что в выражение входит именно та величина, которую мы и пытаемся оценить, Z_1/Z_0. Но можно начать с грубой оценки r и использовать получившееся мостиковое распределение для ее итеративного уточнения (Neal, 2005). То есть мы итеративно пересчитываем отношение и на каждой итерации обновляем значение r.

Связанная выборка по значимости. У обоих методов – AIS и мостиковой выборки – есть свои плюсы. Если $D_{\mathrm{KL}}(p_0 \| p_1)$ не слишком велико (поскольку p_0 и p_1 достаточно близки), то мостиковая выборка может оказаться более эффективным способом оценивания отношения статистических сумм, чем AIS. Если же два распределения отстоят слишком далеко друг от друга, чтобы для наведения моста между ними хватило одного распределения p_*, то для перехода от p_0 к p_1 можно использовать AIS с не-

сколькими промежуточными распределениями. В работе Neal (2005) показано, как метод связанной выборки по значимости может воспользоваться стратегией мостиковой выборки, чтобы связать промежуточные распределения, используемые в AIS, и существенно улучшить общую оценку статистической суммы.

Оценивание статистической суммы на этапе обучения. Хотя AIS принят в качестве стандартного метода оценивания статистической суммы для многих неориентированных моделей, он требует настолько большого объема вычислений, что использовать его на этапе обучения невозможно. Изучались альтернативные стратегии, позволяющие вычислять оценку статистической суммы в процессе обучения.

Применив комбинацию мостиковой выборки, AIS с короткой цепочкой и параллельное темперирование, авторы работы Desjardins et al. (2011) придумали, как прослеживать статистическую сумму ОМБ на всем протяжении обучения. Стратегия основана на хранении независимых оценок статистических сумм ОМБ при каждом значении температуры в схеме параллельного темперирования. Авторы объединили оценки мостиковой выборки отношений статистических сумм соседних цепочек (полученных параллельным темперированием) с оценками AIS в разные моменты времени и в результате получили оценку статистической суммы с низкой дисперсией на каждой итерации обучения.

Описанные в этой главе средства дают много способов преодолеть проблему вычислительно неразрешимых статистических сумм, но в обучении и использовании порождающих моделей есть и другие трудности. Самая главная из них – проблема неразрешимого вывода, к которой мы и переходим.

Глава 19

Приближенный вывод

Многие вероятностные модели сложно обучить из-за трудностей вывода. В контексте глубокого обучения у нас обычно имеется множество наблюдаемых переменных $\boldsymbol{v}$ и множество латентных переменных $\boldsymbol{h}$. Говоря о трудности вывода, мы обычно имеем в виду проблему вычисления $p(\boldsymbol{h} \mid \boldsymbol{v})$ или математических ожиданий относительно этого распределения. Такие операции необходимы, например, при обучении с использованием максимального правдоподобия.

Многие простые графические модели с одним скрытым слоем, например ограниченные машины Больцмана и вероятностный метод главных компонент, определены таким образом, что операции вывода типа вычисления $p(\boldsymbol{h} \mid \boldsymbol{v})$ или математического ожидания выполняются просто. К сожалению, в большинстве графических моделей с несколькими скрытыми слоями апостериорные распределения не поддаются вычислению. В таких моделях любая операция точного вывода требует экспоненциального времени. И эта проблема существует даже в некоторых однослойных моделях, например в модели разреженного кодирования.

В этой главе мы познакомимся с некоторыми приемами преодоления проблемы неразрешимого вывода. А в главе 20 опишем, как они применяются к обучению вероятностных моделей, которые иначе оказались бы неразрешимыми, в т. ч. к глубоким сетям доверия и к глубоким машинам Больцмана.

Неразрешимые проблемы вывода в глубоком обучении обычно возникают из-за взаимодействий между латентными переменными в структурной графической модели. На рис. 19.1 показано несколько примеров. Это могут быть прямые взаимодействия в неориентированных моделях или «оправдывающие» взаимодействия между предками одного и того же видимого блока в ориентированных моделях.

19.1. Вывод как оптимизация

Во многих подходах к решению проблемы трудного вывода используется тот факт, что точный вывод можно описать как задачу оптимизации. Поэтому можно построить приближенный алгоритм вывода, аппроксимируя соответствующую задачу оптимизации.

Для построения задачи оптимизации предположим, что имеется вероятностная модель, содержащая наблюдаемые переменные $\boldsymbol{v}$ и латентные переменные $\boldsymbol{h}$. Мы хотели бы вычислить логарифмическое распределение вероятности наблюдаемых данных $\log p(\boldsymbol{v}; \boldsymbol{\theta})$. Иногда вычислить $\log p(\boldsymbol{v}; \boldsymbol{\theta})$ слишком трудно, если исключение $\boldsymbol{h}$ обходится дорого. Вместо этого мы можем вычислить нижнюю границу $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ величины $\log p(\boldsymbol{v}; \boldsymbol{\theta})$. Эта граница называется **нижней границей свидетельств** (evidence

lower bound – ELBO), а также отрицательной **вариационной свободной энергией**. Точнее говоря, нижняя граница свидетельств определена как

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})), \tag{19.1}$$

где q – произвольное распределение вероятности $\boldsymbol{h}$.

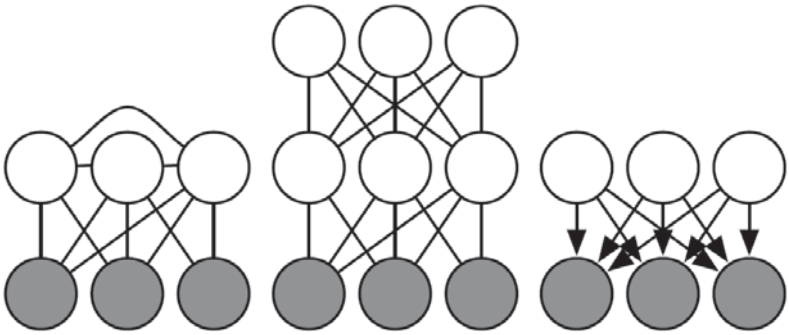

Рис. 19.1 ❖ Неразрешимые проблемы вывода в глубоком обучении обычно являются результатом взаимодействий между латентными переменными в структурной графической модели. Эти взаимодействия могут быть вызваны ребрами, непосредственно соединяющими две латентные переменные, или более длинными путями, которые *активируются*, когда потомок V-структуры – наблюдаемая переменная. (Слева) **Полуограниченная машина Больцмана** (Osindero and Hinton, 2008) со связями между скрытыми блоками. Из-за этих прямых связей между латентными переменными апостериорное распределение неразрешимо, поскольку имеются большие клики латентных переменных. (*В центре*) Для глубокой машины Больцмана, состоящей из слоев переменных без внутрислойных связей, апостериорное распределение все равно неразрешимо из-за связей между слоями. (*Справа*) В этой ориентированной модели взаимодействия между латентными переменными присутствуют, когда видимые переменные наблюдаемые, поскольку каждые две латентные переменные являются родителями одной видимой. В некоторых вероятностных моделях вывод латентных переменных все-таки возможен, несмотря на изображенные на рисунке графические структуры. Так бывает, когда условные распределения вероятности выбраны так, что вносят дополнительные отношения независимости помимо представленных в графе. Например, в вероятностном методе главных компонент структура графа такая, как на правом рисунке, но вывод тем не менее простой благодаря специальным свойствам конкретных условных распределений (линейная комбинация нормальных распределений с взаимно ортогональными базисными векторами)

Поскольку разность между $\log p(\boldsymbol{v})$ и $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ равна расхождению Кульбака–Лейблера, а оно всегда неотрицательно, то $\mathcal{L}$ не может быть больше интересующего нас логарифма вероятности. Равенство достигается тогда и только тогда, когда распределения q и $p(\boldsymbol{h} \mid \boldsymbol{v})$ в точности совпадают.

Как ни странно, для некоторых распределений q вычислить $\mathcal{L}$ гораздо проще. С помощью простых алгебраических преобразований мы можем привести $\mathcal{L}$ к более удобному виду:

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})), \tag{19.2}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{p(\boldsymbol{h} \mid \boldsymbol{v})} \tag{19.3}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{\frac{p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta})}{p(\boldsymbol{v}; \boldsymbol{\theta})}} \tag{19.4}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} [\log q(\boldsymbol{h} \mid \boldsymbol{v}) - \log p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta}) + \log p(\boldsymbol{v}; \boldsymbol{\theta})] \tag{19.5}$$

$$= - \mathbb{E}_{\mathbf{h} \sim q} [\log q(\boldsymbol{h} \mid \boldsymbol{v}) - \log p(\boldsymbol{h}, \boldsymbol{v}; \boldsymbol{\theta})]. \tag{19.6}$$

В результате приходим к каноническому определению нижней границы свидетельств:

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \mathbb{E}_{\mathbf{h} \sim q}[\log p(\boldsymbol{h}, \boldsymbol{v})] + H(q). \tag{19.7}$$

При подходящем выборе q функцию $\mathcal{L}$ можно вычислить. При любом выборе q $\mathcal{L}$ дает нижнюю границу правдоподобия. Чем лучше $q(\boldsymbol{h} \mid \boldsymbol{v})$ аппроксимирует $p(\boldsymbol{h} \mid \boldsymbol{v})$, тем граница $\mathcal{L}$ точнее, т. е. ближе к $\log p(\boldsymbol{v})$. Когда $q(\boldsymbol{h} \mid \boldsymbol{v}) = p(\boldsymbol{h} \mid \boldsymbol{v})$, аппроксимация идеальна и $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \log p(\boldsymbol{h}, \boldsymbol{\theta})$.

Мы можем рассматривать вывод как процедуру нахождения q, доставляющего максимум $\mathcal{L}$. Точный вывод максимизирует $\mathcal{L}$ идеально, т. к. поиск производится во всем семействе функций q, включающем $p(\boldsymbol{h} \mid \boldsymbol{v})$. В этой главе мы продемонстрируем различные виды приближенного вывода с использованием приближенной оптимизации для нахождения q. Мы можем сделать процедуру оптимизации менее дорогой, но приближенной, ограничив семейство распределений q, в котором разрешено производить поиск, или применяя неточную процедуру оптимизации, которая, возможно, не находит истинного максимума функции $\mathcal{L}$, а просто значительно увеличивает ее.

При любом выборе q функция $\mathcal{L}$ остается нижней границей. Граница может быть более или менее точной, более или менее дешевой для вычисления в зависимости от того, какой выбрать подход к задаче оптимизации. Можно получить плохую аппроксимацию q, но уменьшить вычислительную стоимость, воспользовавшись либо неточной процедурой оптимизации, либо точной процедурой, но по ограниченному семейству распределений q.

19.2. EM-алгоритм

Первый основанный на максимизации нижней границы $\mathcal{L}$ алгоритм, который мы рассмотрим, – это популярный EM-алгоритм (expectation maximization) обучения моделей с латентными переменными. Здесь мы опишем его интерпретацию, предложенную в работе Neal and Hinton (1999). В отличие от большинства других алгоритмов в этой главе, EM – это подход не столько к приближенному выводу, сколько к обучению с приближенным апостериорным распределением.

EM-алгоритм состоит из двух чередующихся шагов, повторяемых до достижения сходимости.

- **E-шаг** (вычисление математического ожидания). Обозначим $\boldsymbol{\theta}^{(0)}$ значение параметров в начале шага. Положим $q(\boldsymbol{h}^{(i)} \mid \boldsymbol{v}) = p(\boldsymbol{h}^{(i)} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)})$ для всех индексов i примеров $\boldsymbol{v}^{(i)}$, на которых мы производим обучение (допустимы оба варианта: пакетный и мини-пакетный). Таким образом, q определено в терминах *теку-*

щего значения параметра $\boldsymbol{\theta}^{(0)}$; если мы изменим $\boldsymbol{\theta}$, то $p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})$ изменится, но $q(\boldsymbol{h} \mid \boldsymbol{v})$ останется равным $= p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}^{(0)})$.

- **М-шаг** (максимизация). Полностью или частично максимизировать выражение

$$\sum_i \mathcal{L}(\boldsymbol{v}^{(i)}, \boldsymbol{\theta}, q) \tag{19.8}$$

относительно $\boldsymbol{\theta}$, воспользовавшись любым удобным алгоритмом.

Это можно рассматривать как алгоритм покоординатного восхождения для максимизации $\mathcal{L}$. На одном шаге мы максимизируем $\mathcal{L}$ относительно q, а на другом – $\mathcal{L}$ относительно $\boldsymbol{\theta}$.

Стохастическое градиентное восхождение в моделях с латентными переменными можно рассматривать как частный случай EM-алгоритма, в котором М-шаг состоит из одного шага вычисления градиента. В других вариантах EM-алгоритма таких шагов может быть гораздо больше. Для некоторых семейств моделей М-шаг можно даже выполнять аналитически, находя оптимальное $\boldsymbol{\theta}$ при текущем q.

Несмотря на то что Е-шаг включает точный вывод, мы можем считать, что в EM-алгоритме в некотором смысле используется приближенный вывод. Точнее говоря, на М-шаге предполагается, что одно и то же значение q можно использовать для всех значений $\boldsymbol{\theta}$. Это порождает разрыв между $\mathcal{L}$ и истинным $\log p(\boldsymbol{v})$, по мере того как М-шаг все дальше уходит от значения $\boldsymbol{\theta}^{(0)}$ на Е-шаге. По счастью, Е-шаг снова сократит этот разрыв до нуля при следующем входе в цикл.

EM-алгоритм содержит несколько разных идей. Прежде всего присутствует базовая структура процесса обучения, согласно которой мы обновляем параметры модели с целью повысить правдоподобие пополненного набора данных, в котором все отсутствующие переменные получили значения, предлагаемые оценкой апостериорного распределения. Эта идея встречается не только в EM-алгоритме. Например, применение градиентного спуска для максимизации логарифмического правдоподобия обладает тем же свойством; для вычисления градиента логарифмического правдоподобия нужно находить математические ожидания относительно апостериорного распределения скрытых блоков. Еще одна важная идея EM-алгоритма – то, что мы можем продолжать использовать одно значение q даже после того, как перешли к другому значению $\boldsymbol{\theta}$. Эта идея повсеместно используется в классическом машинном обучении для обновлений с большим М-шагом. В контексте глубокого обучения модели, как правило, слишком сложны, чтобы можно было найти оптимальное обновление с большим М-шагом, так что эта вторая идея, в большей степени относящаяся именно к EM-алгоритму, используется редко.

19.3. MAP-вывод и разреженное кодирование

Обычно термин *вывод* относится к вычислению распределения вероятности одного множества переменных при условии другого множества. При обучении вероятностных моделей с латентными переменными нас обычно интересует вычисление $p(\boldsymbol{h} \mid \boldsymbol{v})$. Альтернативная форма вывода заключается в вычислении одного самого вероятного значения отсутствующих переменных, а не вывода всего распределения их возможных значений. В контексте моделей с латентными переменными это означает, что нужно вычислить

$$\boldsymbol{h}^* = \arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}). \tag{19.9}$$

Это называется выводом **апостериорного максимума**, или MAP-выводом.

MAP-вывод обычно не считается приближенным выводом, т. к. он сводится к точному вычислению самого вероятного значения $\boldsymbol{h}^*$. Однако если мы хотим разработать процесс обучения, основанный на максимизации $\mathcal{L}(\boldsymbol{v}, \boldsymbol{h}, q)$, то полезно рассматривать MAP-вывод как процедуру, поставляющую значение q. В этом смысле можно интерпретировать MAP-вывод как приближенный вывод, поскольку он не дает оптимального q.

Напомним (см. раздел 19.1), что точный вывод заключается в максимизации функции

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) = \mathbb{E}_{\mathbf{h}\sim q}[\log p(\boldsymbol{h}, \boldsymbol{v})] + \mathrm{H}(q) \tag{19.10}$$

относительно q на неограниченном семействе распределений вероятности с применением точного алгоритма оптимизации. Мы можем определить MAP-вывод как вид приближенного вывода, ограничив семейство распределений, из которого выбирается q. Точнее говоря, потребуем, чтобы q было распределением Дирака:

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \delta(\boldsymbol{h} - \boldsymbol{\mu}). \tag{19.11}$$

Это означает, что теперь мы можем управлять q с помощью одного лишь параметра $\boldsymbol{\mu}$. Если опустить члены $\mathcal{L}$, не зависящие от $\boldsymbol{\mu}$, то остается такая задача оптимизации:

$$\boldsymbol{\mu}^* = \arg\max_{\boldsymbol{\mu}} \log p(\boldsymbol{h} = \boldsymbol{\mu}, \boldsymbol{v}), \tag{19.13}$$

эквивалентная задаче MAP-вывода

$$\boldsymbol{h}^* = \arg\max_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}). \tag{19.13}$$

Таким образом, мы можем обосновать процедуру обучения, похожую на EM-алгоритм, в которой MAP-вывод $\boldsymbol{h}^*$ чередуется с обновлением $\boldsymbol{\theta}$ с целью увеличения $\log p(\boldsymbol{h}^*, \boldsymbol{v})$. Как и EM-алгоритм, это вариант покоординатного спуска по $\mathcal{L}$, когда использование вывода для оптимизации $\mathcal{L}$ относительно q чередуется с обновлением параметров для оптимизации $\mathcal{L}$ относительно $\boldsymbol{\theta}$. В обоснование процедуры в целом можно сослаться на тот факт, что $\mathcal{L}$ является нижней границей $\log p(\boldsymbol{v})$. В случае MAP-вывода это обоснование бессодержательно, т. к. граница бесконечно неточная из-за того, что дифференциальная энтропия распределения Дирака равна минус бесконечности. Но прибавление шума к $\boldsymbol{\mu}$ снова делает границу осмысленной.

MAP-вывод обычно используется в глубоком обучении для выделения признаков и в качестве механизма обучения. Основное применение он находит в моделях разреженного кодирования. Напомним (раздел 13.4), что разреженное кодирование – это линейная факторная модель с априорным распределением скрытых блоков, индуцирующим разреженность. Чаще всего выбирается факторное априорное распределение Лапласа, для которого

$$p(h_i) = (\lambda/2)e^{-\lambda|h_i|}. \tag{19.14}$$

Затем видимые блоки порождаются путем выполнения линейного преобразования и прибавления шума:

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}, \beta^{-1}\boldsymbol{I}). \tag{19.15}$$

Вычислить или хотя бы представить $p(\boldsymbol{h} \mid \boldsymbol{v})$ трудно. Любые две переменные h_i и h_j являются родителями $\boldsymbol{v}$. Это означает, что если $\boldsymbol{v}$ наблюдаемая, то графическая модель содержит активный путь, соединяющий h_i и h_j. Следовательно, все скрытые блоки принадлежат одной большой клике в $p(\boldsymbol{h} \mid \boldsymbol{v})$. Если бы модель была гауссовой, то эти взаимодействия можно было бы эффективно смоделировать с помощью ковариационной матрицы, но из-за разреженности априорного распределения взаимодействия гауссовыми не являются.

Поскольку $p(\boldsymbol{h} \mid \boldsymbol{v})$ вычислительно неразрешима, то таковы же логарифмическое правдоподобие и его градиент. Поэтому мы не можем воспользоваться точным обучением с критерием максимального правдоподобия. Вместо этого мы применим MAP-вывод и будем обучать параметры путем максимизации ELBO, определяемой распределением Дирака вокруг MAP-оценки $\boldsymbol{h}$.

Если собрать все векторы $\boldsymbol{h}$ из обучающего набора в матрицу $\boldsymbol{H}$, а все векторы $\boldsymbol{v}$ – в матрицу $\boldsymbol{V}$, то процесс обучения модели разреженного кодирования сведется к минимизации функции

$$J(\boldsymbol{H}, \boldsymbol{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (\boldsymbol{V} - \boldsymbol{H}\boldsymbol{W}^{\top})^2_{i,j}. \tag{19.16}$$

В большинстве применений разреженного кодирования участвует также снижение весов или ограничение на нормы столбцов $\boldsymbol{W}$, чтобы предотвратить патологическое решение с очень малой $\boldsymbol{H}$ и большой $\boldsymbol{W}$.

Для минимизации $\boldsymbol{J}$ мы можем чередовать минимизацию относительно $\boldsymbol{H}$ с минимизацией относительно $\boldsymbol{W}$. Обе подзадачи выпуклые. На самом деле минимизация относительно $\boldsymbol{W}$ – просто задача линейной регрессии. Но минимизация $\boldsymbol{J}$ относительно обоих аргументов обычно не является выпуклой задачей.

Для минимизации относительно $\boldsymbol{H}$ требуются специальные алгоритмы, например алгоритм поиска знака признака (Lee et al., 2007).

19.4. Вариационный вывод и обучение

Мы видели, что нижняя граница свидетельств $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ является нижней границей $\log p(\boldsymbol{v}; \boldsymbol{\theta})$, что вывод можно рассматривать как максимизацию $\mathcal{L}$ относительно q, а обучение – как максимизацию $\mathcal{L}$ относительно $\boldsymbol{\theta}$. Мы видели, что EM-алгоритм позволяет делать большие шаги обучения с фиксированным q и что алгоритмы обучения, основанные на MAP-выводе, дают возможность обучать модель с применением точечной оценки $p(\boldsymbol{h} \mid \boldsymbol{v})$, не выводя всего распределения целиком. Теперь разработаем более общий подход к вариационному обучению.

Основная идея вариационного обучения заключается в том, чтобы максимизировать $\mathcal{L}$ на ограниченном семействе распределений q. Это семейство следует выбирать так, чтобы было легко вычислить $\mathbb{E}_q \log p(\boldsymbol{h} \mid \boldsymbol{v})$. Обычно для этого вводятся предположения о способе представления q в виде произведения.

Типичный подход к вариационному обучению – потребовать, чтобы q было факторным распределением:

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i q(h_i \mid \boldsymbol{v}). \tag{19.17}$$

Это так называемый подход **среднего поля**. В общем случае мы можем наложить на q структуру произвольной графической модели, чтобы гибко определить, сколько взаимодействий должна улавливать наша аппроксимация. Такой общий подход называется **структурным вариационным выводом** (Saul and Jordan, 1996).

Элегантность вариационного подхода состоит в том, что не нужно задавать конкретную параметрическую форму q. Мы описываем, как оно должно факторизоваться, но затем задача оптимизации сама определяет оптимальное распределение вероятности, удовлетворяющее этим ограничениям на факторизацию. Для дискретных латентных переменных это просто означает, что мы пользуемся традиционными методами оптимизации конечного числа переменных, описывающих распределение q. Для непрерывных латентных переменных это означает использование раздела математики, называемого вариационным исчислением, для оптимизации на пространстве функций, в результате чего определяется, какой функцией представлять q. Именно вариационное исчисление стоит за терминами «вариационное обучение» и «вариационный вывод», хотя они применяются и в том случае, когда латентные переменные дискретны и вариационное исчисление ни при чем. В случае непрерывных латентных переменных вариационное исчисление оказывается мощной техникой, снимающей большую часть забот с проектировщика модели, который теперь должен только задать вид факторизации q, а не гадать, как спроектировать конкретное q, способное верно аппроксимировать апостериорное распределение.

Поскольку $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ определена как $\log p(\boldsymbol{v}; \boldsymbol{\theta}) - D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}))$, то максимизацию $\mathcal{L}$ относительно q можно интерпретировать как минимизацию $D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}))$. В этом смысле мы подгоняем q к p. Однако при этом направление расхождения Кульбака–Лейблера противоположно тому, к которому мы привыкли при подгонке аппроксимации. Если для подгонки модели к данным применяется критерий максимального правдоподобия, то мы минимизируем $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$. Как показано на рис. 3.6, это означает, что максимальное правдоподобие поощряет модель иметь высокую вероятность всюду, где высока вероятность данных, тогда как наша основанная на оптимизации процедура вывода поощряет q иметь низкую вероятность там, где низка вероятность апостериорного распределения. У обоих направлений расхождения Кульбака–Лейблера есть желательные и нежелательные свойства. Какое из них использовать, зависит от того, какие свойства приоритетны для приложения. В задаче оптимизации вывода мы выбрали $D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}))$ из соображений удобства вычислений. Точнее говоря, для вычисления $D_{\mathrm{KL}}(q(\boldsymbol{h} \mid \boldsymbol{v}) \| p(\boldsymbol{h} \mid \boldsymbol{v}))$ нужно вычислять математические ожидания относительно q, поэтому, проектируя простое q, мы можем упростить вычисление математических ожиданий. При противоположном направлении расхождения КЛ понадобилось бы вычислять математические ожидания относительно истинного апостериорного распределения. Поскольку форма этого распределения определяется выбором модели, мы не можем точно спроектировать способ вычисления $D_{\mathrm{KL}}(p(\boldsymbol{h} \mid \boldsymbol{v}) \| q(\boldsymbol{h} \mid \boldsymbol{v}))$, характеризующийся низкой стоимостью.

19.4.1. Дискретные латентные переменные

Вариационный вывод с дискретными латентными переменными относительно прост. Мы определяем распределение q, обычно такое, в котором каждый фактор описывается справочной таблицей дискретных состояний. В простейшем случае вектор $\boldsymbol{h}$ бинарный, и мы можем принять предположение среднего поля, согласно которому

q является произведением отдельных h_i. В таком случае можно параметризовать q вектором $\hat{\boldsymbol{h}}$, элементами которого являются вероятности. Тогда $q(h_i = 1 \mid \boldsymbol{v}) = \hat{h}_i$.

Решив, как представлять q, мы затем просто оптимизируем параметры. Если латентные переменные дискретны, то это стандартная задача оптимизации. В принципе, для выбора q можно было бы применить любой алгоритм оптимизации, например градиентный спуск.

Поскольку эта оптимизация производится во внутреннем цикле алгоритма обучения, она должна быть очень быстрой. Для достижения нужной скорости обычно применяются специальные алгоритмы для решения сравнительно небольших и простых задач за малое число итераций. Распространенный выбор – итерационный поиск неподвижной точки, т. е. решение уравнений вида

$$\frac{\partial}{\partial \hat{h}_i}\mathcal{L} = 0 \tag{19.18}$$

относительно $\hat{h}_i$. Мы повторно обновляем различные элементы $\hat{\boldsymbol{h}}$, пока не будет удовлетворен критерий сходимости.

Для конкретности покажем, как вариационный вывод применяется к **модели бинарного разреженного кодирования** (мы используем модель из работы Henniges et al. [2010], но демонстрируем традиционный общий подход на основе среднего поля, тогда как авторы разработали специализированный алгоритм). В выводе довольно много математических деталей, он предназначен для читателей, которые хотят разрешить все неоднозначности приведенного выше высокоуровневого описания вариационного вывода и обучения. Читатели, не планирующие разрабатывать или реализовывать алгоритмы вариационного обучения, могут без ущерба для понимания сразу перейти к следующему разделу. Тем же, кто решил разобраться в примере бинарного разреженного кодирования, мы рекомендуем освежить в памяти полезные свойства функций, часто встречающихся в вероятностных моделях (см. раздел 3.10). Мы будем пользоваться ими без дальнейших оговорок.

В модели бинарного разреженного кодирования вход $\boldsymbol{v} \in \mathbb{R}^n$ генерируется по модели посредством прибавления гауссова шума к сумме m различных компонент, каждая из которых может присутствовать или отсутствовать. Каждая компонента включается или выключается соответствующим скрытым блоком в $\boldsymbol{h} \in \{0, 1\}^m$:

$$p(h_i = 1) = \sigma(b_i), \tag{19.19}$$

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1}), \tag{19.20}$$

где $\boldsymbol{b}$ – обучаемый вектор смещений, $\boldsymbol{W}$ – обучаемая матрица весов, а $\boldsymbol{\beta}$ – обучаемая диагональная матрица точности.

Для обучения этой модели с критерием максимального правдоподобия необходимо брать производные по параметрам. Рассмотрим производную по одному из смещений:

$$\frac{\partial}{\partial b_i}\log p(\boldsymbol{v}) \tag{19.21}$$

$$= \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{v})}{p(\boldsymbol{v})} \tag{19.22}$$

$$= \frac{\frac{\partial}{\partial b_i} \sum_{\boldsymbol{h}} p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \tag{19.23}$$

$$= \frac{\frac{\partial}{\partial b_i} \sum_{\boldsymbol{h}} p(\boldsymbol{h}) p(\boldsymbol{v} \mid \boldsymbol{h})}{p(\boldsymbol{v})} \tag{19.24}$$

$$= \frac{\sum_{\boldsymbol{h}} p(\boldsymbol{v} \mid \boldsymbol{h}) \frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{v})} \tag{19.25}$$

$$= \sum_{\boldsymbol{h}} p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{\frac{\partial}{\partial b_i} p(\boldsymbol{h})}{p(\boldsymbol{h})} \tag{19.26}$$

$$= \mathbb{E}_{\mathbf{h} \sim p(\boldsymbol{h} \mid \boldsymbol{v})} \frac{\partial}{\partial b_i} \log p(\boldsymbol{h}). \tag{19.27}$$

Здесь требуется вычислять математические ожидания относительно $p(\boldsymbol{h} \mid \boldsymbol{v})$. К сожалению, $p(\boldsymbol{h} \mid \boldsymbol{v})$ – сложное распределение. На рис. 19.2 показана структура графов $p(\boldsymbol{h}, \boldsymbol{v})$ и $p(\boldsymbol{h} \mid \boldsymbol{v})$. Апостериорному распределению соответствует полный граф скрытых блоков, поэтому алгоритмы исключения переменных не помогут вычислить нужных математических ожиданий быстрее, чем полным перебором.

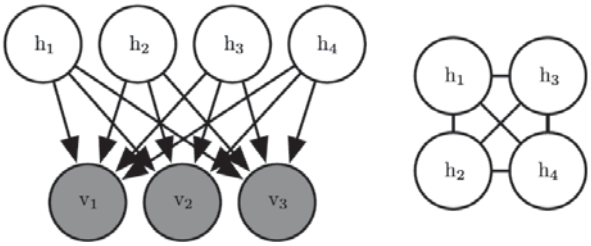

Рис. 19.2 ❖ Структура графа модели бинарного разреженного кодирования с четырьмя скрытыми блоками. (*Слева*) Структура графа $p(\boldsymbol{h}, \boldsymbol{v})$. Отметим, что ребра ориентированные и что два любых скрытых блока являются родителями каждого видимого блока. (*Справа*) Структура графа $p(\boldsymbol{h} \mid \boldsymbol{v})$. Чтобы учесть активные пути между сородителями, в апостериорном распределении должно быть проведено ребро между любыми двумя скрытыми блоками

Эту трудность можно разрешить, воспользовавшись вариационным выводом и вариационным обучением.

Применим аппроксимацию среднего поля:

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i q(h_i \mid \boldsymbol{v}). \tag{19.28}$$

Латентные переменные в модели бинарного разреженного кодирования являются бинарными, поэтому для представления факторного распределения q нам нужно просто смоделировать m распределений Бернулли $q(h_i \mid \boldsymbol{v})$. Средние распределений Бер-

нулли естественно представить вектором вероятностей $\hat{\boldsymbol{h}}$ – таким, что $q(h_i = 1 \mid \boldsymbol{v}) = \hat{h}_i$. Наложим ограничение – $\hat{h}_i$ не может быть равно 0 или 1, – чтобы избежать ошибок при вычислении, например $\log \hat{h}_i$.

Как мы увидим, уравнения вариационного вывода никогда не присваивают $\hat{h}_i$ значение 0 или 1 аналитически. Но в программной реализации из-за ошибок округления такие значения возможны. Программно мы могли бы реализовать двоичное разреженное кодирование, используя неограниченный вектор вариационных параметров $\boldsymbol{z}$, и получить $\hat{\boldsymbol{h}}$ из соотношения $\hat{\boldsymbol{h}} = \sigma(\boldsymbol{z})$. Тогда можно безопасно вычислять $\log \hat{h}_i$, применяя тождество $\log \sigma(z_i) = -\zeta(-z_i)$, связывающее сигмоиду с функцией softplus.

Приступая к выводу уравнений вариационного обучения в модели бинарного разреженного кодирования, покажем, что благодаря использованию аппроксимации среднего поля обучение становится вычислительно разрешимым.

Нижняя граница свидетельств имеет вид:

$$\mathcal{L}(\boldsymbol{v}; \boldsymbol{\theta}; q) \tag{19.29}$$

$$= \mathbb{E}_{\mathbf{h}\sim q}[\log p(\boldsymbol{h}, \boldsymbol{v})] + H(q) \tag{19.30}$$

$$= \mathbb{E}_{\mathbf{h}\sim q}[\log p(\boldsymbol{h}) + \log p(\boldsymbol{v} \mid \boldsymbol{h}) - \log q(\boldsymbol{h} \mid \boldsymbol{v})] \tag{19.31}$$

$$= \mathbb{E}_{\mathbf{h}\sim q}\left[\sum_{i=1}^{m} \log p(h_i) + \sum_{i=1}^{n} \log p(v_i \mid \boldsymbol{h}) - \sum_{i=1}^{m} \log q(h_i \mid \boldsymbol{v})\right] \tag{19.32}$$

$$= \sum_{i=1}^{m} [\hat{h}_i(\log \sigma(b_i) - \log \hat{h}_i) + (1 - \hat{h}_i)(\log \sigma(-b_i) - \log(1 - \hat{h}_i))] \tag{19.33}$$

$$+ \mathbb{E}_{\mathbf{h}\sim q}\left[\sum_{i=1}^{m} \log \sqrt{\frac{\beta_i}{2\pi}} \exp\left(-\frac{\beta_i}{2}(v_i - \boldsymbol{W}_{i,:}\boldsymbol{h})^2\right)\right] \tag{19.34}$$

$$= \sum_{i=1}^{m} [\hat{h}_i(\log \sigma(b_i) - \log \hat{h}_i) + (1 - \hat{h}_i)(\log \sigma(-b_i) - \log(1 - \hat{h}_i))] \tag{19.35}$$

$$+ \frac{1}{2}\sum_{i=1}^{n}\left[\log \frac{\beta_i}{2\pi} - \beta_i\left(v_i^2 - 2v_i \boldsymbol{W}_{i,:}\hat{\boldsymbol{h}} + \sum_j \left[W_{i,j}^2 \hat{h}_j + \sum_{k\neq j} W_{i,j} W_{i,k} \hat{h}_j \hat{h}_k\right]\right)\right]. \tag{19.36}$$

Выглядят эти уравнения не очень эстетично, но показывают, что $\mathcal{L}$ можно выразить с помощью небольшого числа простых арифметических операций. Поэтому нижняя граница свидетельств $\mathcal{L}$ разрешима. Мы можем использовать $\mathcal{L}$ вместо неразрешимого логарифмического правдоподобия.

В принципе, можно было бы просто выполнить градиентное восхождение по $\boldsymbol{v}$ и $\boldsymbol{h}$, получив тем самым вполне приемлемую комбинацию алгоритмов вывода и обучения. Но обычно так не поступают по двум причинам. Во-первых, понадобилось бы хранить $\hat{\boldsymbol{h}}$ для каждого $\boldsymbol{v}$, а мы все же предпочитаем алгоритмы, не требующие выделения памяти для каждого примера. Трудно масштабировать алгоритмы обучения на миллиарды примеров, если для каждого необходимо хранить динамически обновляемый вектор. Во-вторых, хотелось бы очень быстро выделять признаки $\hat{\boldsymbol{h}}$, чтобы распознавать содержимое $\boldsymbol{v}$. В настоящей развернутой системе требуется вычислять $\hat{\boldsymbol{h}}$ в режиме реального времени.

По этим причинам градиентное восхождение обычно не применяется для вычисления параметров среднего поля $\hat{\boldsymbol{h}}$. Вместо этого мы производим их быструю оценку с помощью уравнений неподвижной точки.

Идея этих уравнений – в том, что мы ищем локальный максимум относительно $\hat{\boldsymbol{h}}$, где $\nabla_{\boldsymbol{h}} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) = 0$. Мы не можем эффективно решить это уравнение относительно всех компонент $\hat{\boldsymbol{h}}$ одновременно. Но его можно решить для одной переменной:

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) = 0. \tag{19.37}$$

Таким образом, мы можем итеративно решать уравнение для $i = 1, ..., m$ и повторять цикл, пока не будет удовлетворен критерий сходимости. Типичный критерий говорит, что нужно остановиться, когда по завершении полного цикла обновления $\mathcal{L}$ улучшается не больше, чем на заданную величину, или когда $\hat{\boldsymbol{h}}$ изменяется не больше, чем на заданную величину.

Итерационное решение уравнений неподвижной точки среднего поля – это общий прием, который может приводить к быстрому вариационному выводу для широкого класса моделей. Для конкретности покажем, как все это выглядит в случае модели бинарного разреженного кодирования. Прежде всего необходимо выписать выражение для производных по $\hat{h}_i$. Для этого подставим выражение (19.36) в левую часть уравнения (19.37):

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \hat{\boldsymbol{h}}) \tag{19.38}$$

$$= \frac{\partial}{\partial \hat{h}_i} \left[\sum_{j=1}^{m} [\hat{h}_j (\log \sigma(b_j) - \log \hat{h}_j) + (1 - \hat{h}_j)(\log \sigma(-b_j) - \log(1 - \hat{h}_j))] \right] \tag{19.39}$$

$$+ \frac{1}{2} \sum_{j=1}^{n} \left[\log \frac{\beta_j}{2\pi} - \beta_j \left(v_j^2 - 2 v_j \boldsymbol{W}_{j,:} \hat{\boldsymbol{h}} + \sum_k \left[W_{j,k}^2 \hat{h}_k + \sum_{l \neq k} W_{j,k} W_{j,l} \hat{h}_k \hat{h}_l \right] \right) \right] \tag{19.40}$$

$$= \log \sigma(b_i) - \log \hat{h}_i - 1 + \log(1 - \hat{h}_i) + 1 - \log \sigma(-b_i) \tag{19.41}$$

$$+ \sum_{j=1}^{n} \left[\beta_j \left(v_j W_{j,i} - \frac{1}{2} W_{j,i}^2 - \sum_{k \neq i} \boldsymbol{W}_{j,k} \boldsymbol{W}_{j,i} \hat{h}_k \right) \right] \tag{19.42}$$

$$= b_i - \log \hat{h}_i + \log(1 - \hat{h}_i) + \boldsymbol{v}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \frac{1}{2} \boldsymbol{W}_{:,i}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \sum_{j \neq i} \boldsymbol{W}_{:,j}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} \hat{h}_j. \tag{19.43}$$

Чтобы применить правило обновления уравнений неподвижной точки, находим $\hat{h}_i$, обращающее выражение (19.43) в 0:

$$\hat{h}_i = \sigma \left(b_i + \boldsymbol{v}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \frac{1}{2} \boldsymbol{W}_{:,i}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \sum_{j \neq i} \boldsymbol{W}_{:,j}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} \hat{h}_j \right). \tag{19.44}$$

Теперь мы видим, что существует тесная связь между рекуррентными нейронными сетями и выводом в графических моделях. Точнее говоря, уравнения неподвижной точки среднего поля определяют рекуррентную нейронную сеть, задача которой – выполнение вывода. Мы показали, как вывести эту сеть из описания модели, но можно также обучать сеть вывода непосредственно. В главе 20 изложено несколько идей, относящихся к этой теме.

В случае бинарного разреженного кодирования мы видим, что связь в рекуррентной сети, описываемой уравнением (19.44), заключается в повторном обновлении

скрытых блоков на основе изменения значений соседних скрытых блоков. Вход всегда отправляет фиксированное сообщение $\boldsymbol{v}^\top \boldsymbol{\beta} \boldsymbol{W}$ скрытым блокам, но скрытые блоки постоянно обновляют сообщения, которыми обмениваются между собой. Точнее говоря, два блока $\hat{h}_i$ и $\hat{h}_j$ тормозят друг друга, когда их векторы весов совмещены. Это форма конкуренции – из двух скрытых блоков, объясняющих вход, активным разрешено оставаться только тому, который дает лучшее объяснение. Наличие такой конкуренции – это попытка аппроксимации среднего поля уловить оправдывающие взаимодействия в апостериорном распределении бинарного разреженного кодирования. На самом деле эффект оправдания должен приводить к многомодальному апостериорному распределению, поэтому если мы будем выбирать примеры из апостериорного распределения, то в одних будет активен один блок, в других – другой, но найдется очень мало примеров, в которых активны оба блока. К сожалению, оправдывающие взаимодействия невозможно смоделировать факторным распределением q, используемым в случае среднего поля, поэтому аппроксимация среднего поля вынуждена назначать модели одну моду. Это пример поведения, показанного на рис. 3.6.

Мы можем переписать уравнение (19.44) в эквивалентной форме, позволяющей прийти к дополнительным заключениям:

$$\hat{h}_i = \sigma\left(b_i + \left(\boldsymbol{v} - \sum_{j \neq i} \boldsymbol{W}_{:,j} \hat{h}_j \right)^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} - \frac{1}{2} \boldsymbol{W}_{:,i}^\top \boldsymbol{\beta} \boldsymbol{W}_{:,i} \right). \tag{19.45}$$

В этой формулировке видно, что вход на каждом шаге состоит из $\boldsymbol{v} - \Sigma_{j \neq i} \boldsymbol{W}_{:,j} \hat{h}_i$, а не $\boldsymbol{v}$. Поэтому можно считать, что блок i пытается закодировать остаточную ошибку в $\boldsymbol{v}$ при известных кодах других блоков. Таким образом, разреженное кодирование можно интерпретировать как итеративный автокодировщик, который повторно кодирует и декодирует свой вход, пытаясь исправить ошибки реконструкции после каждой итерации.

В этом примере мы вывели правило обновления по одному блоку за раз. Было бы хорошо одновременно обновлять несколько блоков. Некоторые графические модели, в т. ч. глубокие машины Больцмана, структурированы так, что можно одновременно находить много элементов $\hat{\boldsymbol{h}}$. К сожалению, бинарное разреженное кодирование не допускает такого блочного обновления. Вместо этого для блочного обновления можно использовать эвристическую технику **демпфирования** (damping). Смысл ее состоит в том, что мы находим из уравнений индивидуально оптимальные значения каждого элемента $\hat{\boldsymbol{h}}$, а затем производим малый сдвиг всех значений в этом направлении. Не гарантируется, что при этом $\mathcal{L}$ на каждом шаге будет увеличиваться, но для многих моделей этот подход на практике дает хорошие результаты. Дополнительные сведения о выборе уровня синхронности и стратегиях демпфирования в алгоритмах передачи сообщений см. в работе Koller and Friedman (2009).

19.4.2. Вариационное исчисление

Прежде чем продолжить экскурс в вариационное обучение, необходимо краткое введение в важный математический инструментарий: **вариационное исчисление**.

Многие методы машинного обучения основаны на минимизации функции $J(\boldsymbol{\theta})$, т. е. нахождении входного вектора $\boldsymbol{\theta} \in \mathbb{R}^n$, доставляющего ей минимальное значение. Для этого можно применить методы многомерного математического анализа и линейной алгебры к нахождению критических точек, в которых $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbf{0}$. Но в неко-

торых случаях нам на самом деле нужно найти функцию $f(\boldsymbol{x})$, например если ищется функция плотности вероятности некоторой случайной величины. Именно это и позволяет сделать вариационное исчисление.

Функция от функции f называется **функционалом** $J[f]$. Точно так же, как мы берем частные производные функции по элементам ее векторного аргумента, можно брать и **функциональные производные**, называемые также **вариационными производными** функционала $J[f]$ по значениям функции $f(\boldsymbol{x})$ в любой точке $\boldsymbol{x}$. Функциональная производная функционала J по значению функции f в точке $\boldsymbol{x}$ обозначается $(\delta/\delta f(x))J$.

Полная формальная разработка понятия функциональных производных выходит за рамки этой книги. Нам достаточно знать, что для дифференцируемой функции $f(\boldsymbol{x})$ и дифференцируемой функции $g(y, \boldsymbol{x})$ с непрерывными производными справедливо тождество

$$\frac{\delta}{\delta f(\boldsymbol{x})}\int g(f(\boldsymbol{x}), \boldsymbol{x})d\boldsymbol{x} = \frac{\partial}{\partial y}g(f(\boldsymbol{x}), \boldsymbol{x}). \tag{19.46}$$

Чтобы понять интуитивный смысл этого тождества, представим себе, что $f(\boldsymbol{x})$ – вектор с несчетным множеством элементов, индексированный вещественным вектором $\boldsymbol{x}$. При таком (не вполне полном) взгляде приведенное выше тождество не отличается от того, что мы имели бы для вектора $\boldsymbol{\theta} \in \mathbb{R}^n$, индексированного положительными целыми числами:

$$\frac{\partial}{\partial \theta_i}\sum_j g(\theta_j, j) = \frac{\partial}{\partial \theta_i}g(\theta_i, i). \tag{19.47}$$

Многие результаты в литературе по машинному обучению изложены в терминах более общего **уравнения Эйлера–Лагранжа**, в котором g может зависеть не только от значения f, но и от производных f, но нам такая общая форма не понадобится.

Для оптимизации функции относительно вектора мы вычисляем градиент этой функции по вектору, приравниваем все элементы градиента к нулю и решаем получившуюся систему уравнений. Точно так же для оптимизации функционала следует искать функцию из системы уравнений, выражающей равенство нулю функциональных производных в каждой точке.

В качестве примера рассмотрим задачу о нахождении функции распределения вероятности от $x \in \mathbb{R}$ с минимальной дифференциальной энтропией. Напомним, что энтропия распределения вероятности $p(x)$ определяется формулой

$$H[p] = -\mathbb{E}_x \log p(x). \tag{19.48}$$

В непрерывном случае математическое ожидание – это интеграл:

$$H[p] = -\int p(x)\log p(x)dx. \tag{19.49}$$

Мы не можем просто максимизировать $H[p]$ относительно функции $p(x)$, потому что результатом может оказаться функция, не являющаяся распределением вероятности. Поэтому нам придется воспользоваться множителями Лагранжа, чтобы добавить ограничение: интеграл $p(x)$ должен быть равен 1. Кроме того, энтропия должна неограниченно возрастать с ростом дисперсии. Из-за этого вопрос о распределении с максимальной энтропией становится неинтересным. Вместо него зададимся вопросом о том, какое распределение имеет наибольшую энтропию при фиксирован-

ной дисперсии σ^2. Наконец, задача недетерминированная, потому что распределение можно произвольно сдвинуть, не меняя энтропию. Чтобы получить единственное решение, добавим еще ограничение, что среднее значение распределения должно быть равно μ. Функционал Лагранжа для этой задачи оптимизации имеет вид

$$\mathcal{L}[p] = \lambda_1(\int p(x)dx - 1) + \lambda_2(\mathbb{E}[x] - \mu) + \lambda_3(\mathbb{E}[(x-\mu)^2] - \sigma^2) + H[p] \tag{19.50}$$

$$= \int(\lambda_1 p(x) + \lambda_2 p(x)x + \lambda_3 p(x)(x-\mu)^2 - p(x)\log p(x))dx - \lambda_1 - \mu\lambda_2 - \sigma^2\lambda_3. \tag{19.51}$$

Для минимизации лагранжиана относительно p приравняем функциональные производные к нулю:

$$\forall x, \frac{\delta}{\delta p(x)}\mathcal{L} = \lambda_1 + \lambda_2 x + \lambda_3(x-\mu)^2 - 1 - \log p(x) = 0. \tag{19.52}$$

Это условие сообщает нам о функциональной форме $p(x)$. После простых алгебраических преобразований получаем

$$p(x) = \exp(\lambda_1 + \lambda_2 x + \lambda_3(x-\mu)^2 - 1). \tag{19.53}$$

Мы нигде не предполагали, что функциональная форма $p(x)$ именно такова, это выражение получилось в результате аналитической минимизации функционала. Чтобы довести до конца решение задачи минимизации, необходимо выбрать такие значения λ, при которых удовлетворяются все ограничения. Мы вольны выбирать любые значения λ, т. к. градиент лагранжиана по переменным λ равен 0, коль скоро удовлетворяются ограничения. Чтобы удовлетворить все ограничения, положим $\lambda_1 = 1 - \log\sigma\sqrt{2\pi}$, $\lambda_2 = 0$, $\lambda_3 = -1/(2\sigma^2)$. Тогда получится

$$p(x) = \mathcal{N}(x; \mu, \sigma^2). \tag{19.54}$$

Это одна из причин использования нормального распределения в случае, когда истинное неизвестно. Поскольку энтропия нормального распределения максимальна, такое предположение накладывает наименее строгую структуру.

Исследуя критические точки функционала Лагранжа для энтропии, мы нашли только одну такую точку, соответствующую максимуму энтропии при фиксированной дисперсии. А что сказать о функции распределения вероятности, которая минимизирует энтропию? Почему мы не нашли вторую критическую точку, соответствующую минимуму? Причина в том, что не существует функции, доставляющей минимум энтропии. Если увеличивать плотность вероятности в двух точках, $x = \mu + \sigma$ и $x = \mu - \sigma$, уменьшая ее во всех остальных x, то энтропия будет уменьшаться, а дисперсия останется постоянной. Однако для функции, которая сосредоточивает всю массу в двух точках, делая ее равной 0 в остальных, интеграл не равен 1, и она не является допустимым распределением вероятности. Поэтому не существует распределения вероятности с минимальной энтропией, как не существует наименьшего положительного вещественного числа. Мы можем лишь сказать, что существует последовательность распределений вероятности, сходящаяся к концентрации массы всего в двух точках. Этот вырожденный случай можно описать как смесь распределений Дирака. Поскольку распределение Дирака не описывается одной функцией распределения вероятности, то никакая смесь распределений Дирака не соответствует одной конкретной точке в пространстве функций. Такие распределения невидимы нашему методу поиска точек, в которых функциональные производные равны 0. Это

ограничение самого метода. Распределения, подобные дираковскому, следует искать другими методами, например угадать решение и затем доказать его правильность.

19.4.3. Непрерывные латентные переменные

Если графическая модель содержит непрерывные латентные переменные, то мы все равно можем произвести вариационный вывод и обучение путем максимизации $\mathcal{L}$. Однако теперь для максимизации $\mathcal{L}$ относительно $q(\boldsymbol{h} \mid \boldsymbol{v})$ нужно использовать вариационное исчисление.

На практике в большинстве случаев не приходится решать вариационные задачи самостоятельно. Вместо этого имеется общее уравнение для обновления неподвижной точки среднего поля. Если принять аппроксимацию среднего поля

$$q(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i q(h_i \mid \boldsymbol{v}) \tag{19.55}$$

и зафиксировать $q(h_j \mid \boldsymbol{v})$ для всех $j \neq i$, то оптимальное распределение $q(h_i \mid \boldsymbol{v})$ можно получить нормировкой ненормированного распределения

$$\tilde{q}(h_i \mid \boldsymbol{v}) = \exp(\mathbb{E}_{\mathbf{h}_{-i} \sim q(\mathbf{h}_{-i} \mid \boldsymbol{v})} \log \tilde{p}(\boldsymbol{v}, \boldsymbol{h})) \tag{19.56}$$

при условии что p не назначает вероятность 0 ни одной совместной комбинации переменных. Перенос математического ожидания внутрь уравнения дает корректную функциональную форму $q(h_i \mid \boldsymbol{v})$. Непосредственный вывод функциональных форм q методами вариационного исчисления необходим только в случае, когда цель – разработать новый вид вариационного обучения; уравнение (19.56) дает аппроксимацию среднего поля для любой вероятностной модели.

Уравнение (19.56) – это уравнение неподвижной точки, которое следует итеративно применять для каждого значения i до достижения сходимости. Однако этим оно не исчерпывается. Оно сообщает нам функциональную форму оптимального решения вне зависимости от того, найдено оно из уравнения неподвижной точки или иным способом. Это означает, что мы можем взять функциональную форму из этого уравнения, но рассматривать некоторые значения в ней как параметры, которые можно оптимизировать с помощью любого алгоритма по своему выбору.

В качестве примера рассмотрим простую вероятностную модель с латентными переменными $\boldsymbol{h} \in \mathbb{R}^2$ и единственной видимой переменной v. Предположим, что $p(\boldsymbol{h}) = \mathcal{N}(\boldsymbol{h}; 0, \boldsymbol{I})$ и $p(v \mid \boldsymbol{h}) = \mathcal{N}(v; \boldsymbol{w}^\top \boldsymbol{h}; 1)$. Мы могли бы упростить эту модель, исключив $\boldsymbol{h}$ посредством интегрирования; в результате получится просто нормальное распределение $\boldsymbol{v}$. Сама по себе модель не интересна; мы построили ее только ради демонстрации того, как вариационное исчисление применяется к вероятностному моделированию.

Истинное апостериорное распределение с точностью до нормировочной постоянной имеет вид

$$p(\boldsymbol{h} \mid \boldsymbol{v}), \tag{19.57}$$

$$\propto p(\boldsymbol{h}, \boldsymbol{v}), \tag{19.58}$$

$$= p(h_1)p(h_2)p(\boldsymbol{v} \mid \boldsymbol{h}), \tag{19.59}$$

$$\propto \exp\left(-\frac{1}{2}[h_1^2 + h_2^2 + (v - h_1 w_1 - h_2 w_2)^2]\right) \tag{19.60}$$

$$= \exp\left(-\frac{1}{2}[h_1^2 + h_2^2 + v^2 + h_1^2 w_1^2 + h_2^2 w_2^2 - 2vh_1 w_1 - 2vh_2 w_2 + 2h_1 w_1 h_2 w_2] \right). \tag{19.60}$$

Из-за присутствия членов с произведением h_1 и h_2 истинное апостериорное распределение не факторизуется по h_1 и h_2.

Применяя формулу (19.56), находим, что

$$\tilde{q}(h_i \mid \boldsymbol{v}), \tag{19.62}$$

$$= \exp(\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h}_2 \mid \boldsymbol{v})} \log \tilde{p}(\boldsymbol{v}, \boldsymbol{h})), \tag{19.63}$$

$$= \exp\left(-\frac{1}{2} \mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h}_2 \mid \boldsymbol{v})} [h_1^2 + h_2^2 + v^2 + h_1^2 w_1^2 + h_2^2 w_2^2 \right. \tag{19.64}$$

$$\left. - 2vh_1 w_1 - 2vh_2 w_2 + 2h_1 w_1 h_2 w_2] \right). \tag{19.65}$$

Отсюда видно, что нам нужно получить из $q(h_2 \mid \boldsymbol{v})$, по существу, только два значения: $\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h} \mid \boldsymbol{v})}[h_2]$ и $\mathbb{E}_{\mathrm{h}_2 \sim q(\mathrm{h} \mid \boldsymbol{v})}[h_2^2]$. Если обозначить их $\langle h_2 \rangle$ и $\langle h_2^2 \rangle$, то получим

$$\tilde{q}(h_1 \mid \boldsymbol{v}) = \exp\left(-\frac{1}{2}[h_1^2 + \left\langle h_2^2 \right\rangle + v^2 + h_1^2 w_1^2 + \left\langle h_2^2 \right\rangle w_2^2 \right. \tag{19.66}$$

$$\left. - 2vh_1 w_1 - 2v\left\langle h_2 \right\rangle \boldsymbol{w}_2 + 2h_1 w_1 \left\langle h_2 \right\rangle w_2] \right). \tag{19.67}$$

Отсюда следует, что $\tilde{q}$ имеет функциональную форму гауссианы. Следовательно, можно заключить, что $q(\boldsymbol{h} \mid \boldsymbol{v}) = \mathcal{N}(\boldsymbol{h}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1})$, где вектор $\boldsymbol{\mu}$ и диагональная матрица $\boldsymbol{\beta}$ – вариационные параметры, которые можно оптимизировать любым способом. Важно помнить, что мы нигде не предполагали, что q будет нормальным распределением, это получилось автоматически в результате применения вариационного исчисления для максимизации q относительно $\mathcal{L}$. Применив тот же подход к другой модели, мы получили бы другую функциональную форму q.

Конечно, это всего лишь простой пример, сконструированный специально для демонстрации. Примеры реального применения вариационного обучения с непрерывными переменными в контексте глубокого обучения см. в работе Goodfellow et al. (2013d).

19.4.4. Взаимодействия между обучением и выводом

Использование приближенного вывода в составе алгоритма обучения влияет на процесс обучения, а это, в свою очередь, сказывается на верности алгоритма вывода.

Точнее говоря, алгоритм обучения стремится адаптировать модель таким образом, чтобы предположения, лежащие в основе алгоритма приближенного вывода, больше походили на правду. При обучении параметров метод вариационного обучения увеличивает математическое ожидание

$$\mathbb{E}_{\mathbf{h} \sim q} \log p(\boldsymbol{v}, \boldsymbol{h}). \tag{19.68}$$

При данном $\boldsymbol{v}$ это приводит к увеличению $p(\boldsymbol{h} \mid \boldsymbol{v})$ для значений $\boldsymbol{h}$ с высокой вероятностью в распределении $q(\boldsymbol{h} \mid \boldsymbol{v})$ и к уменьшению $p(\boldsymbol{h} \mid \boldsymbol{v})$ для $\boldsymbol{h}$ с низкой вероятностью.

При таком поведении наши предположения, положенные в основу аппроксимации, становятся сбывающимися пророчествами. Если мы обучим модель с унимодальным приближенным апостериорным распределением, то получим модель, для которой ис-

тинное апостериорное распределение гораздо ближе к унимодальному, чем было бы при обучении модели с помощью точного вывода.

Таким образом, вычислить истинный вред, причиняемый модели вариационной аппроксимацией, очень трудно. Существует несколько методов оценивания $\log p(\boldsymbol{v})$. Зачастую мы оцениваем $\log p(\boldsymbol{v}; \boldsymbol{\theta})$ после обучения модели и обнаруживаем, что разрыв с $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q)$ мал. Отсюда мы делаем вывод, что наша вариационная аппроксимация в целом верна или что она причинила небольшой вред процессу обучения. Чтобы измерить истинный вред, привнесенный вариационной аппроксимацией, нам нужно знать $\boldsymbol{\theta}^* = \max_{\boldsymbol{\theta}} \log p(\boldsymbol{v}; \boldsymbol{\theta})$. Может быть так, что соотношения $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, q) \approx \log p(\boldsymbol{v}; \boldsymbol{\theta})$ и $\log p(\boldsymbol{v}; \boldsymbol{\theta}) \ll \log p(\boldsymbol{v}; \boldsymbol{\theta}^*)$ удовлетворяются одновременно. Если $\max_q \mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}^*, q) \ll \log p(\boldsymbol{v}; \boldsymbol{\theta}^*)$ из-за того, что $\boldsymbol{\theta}^*$ индуцирует слишком сложное апостериорное распределение, которое наше семейство q неспособно уловить, то процесс обучения никогда не приблизится к $\boldsymbol{\theta}^*$. Такую проблему очень трудно обнаружить, поскольку узнать наверняка, что это случилось, можно, лишь имея для сравнения другой, более качественный алгоритм обучения, способный найти $\boldsymbol{\theta}^*$.

19.5. Обученный приближенный вывод

Мы видели, что вывод можно рассматривать как процедуру оптимизации, которая увеличивает значение функции $\mathcal{L}$. Явное выполнение оптимизации с помощью таких итеративных процедур, как уравнения неподвижной точки или градиентная оптимизация, часто оказывается делом дорогим и долгим. Поэтому расходов стремятся избежать, обучая модель выполнять приближенный вывод. Точнее говоря, мы можем считать, что процесс оптимизации – это функция f, отображающая вход $\boldsymbol{v}$ в приближенное распределение $q^* = \arg\max_q \mathcal{L}(\boldsymbol{v}, q)$. Коль скоро многошаговая итеративная процедура рассматривается просто как функция, мы можем аппроксимировать ее нейронной сетью, реализующей аппроксимацию $\hat{f}(\boldsymbol{v}, \boldsymbol{\theta})$.

19.5.1. Бодрствование-сон

Одна из основных трудностей при обучении модели выводу $\boldsymbol{h}$ из $\boldsymbol{v}$ состоит в том, что у нас нет помеченного обучающего набора. Мы имеем $\boldsymbol{v}$, но не знаем соответствующего $\boldsymbol{h}$. Отображение $\boldsymbol{v}$ в $\boldsymbol{h}$ зависит от выбора семейства моделей и эволюционирует по мере того, как в процессе обучения изменяется $\boldsymbol{\theta}$. В алгоритме бодрствования-сна (wake-sleep) (Hinton et al., 1995b; Frey et al., 1996) эта проблема решается путем выборки как $\boldsymbol{h}$, так и $\boldsymbol{v}$ из модельного распределения. Например, в ориентированной модели это легко сделать с помощью предковой выборки, начинающейся в $\boldsymbol{h}$ и заканчивающейся в $\boldsymbol{v}$. Тогда сеть вывода можно обучить выполнению обратного отображения: предсказывать, какое $\boldsymbol{h}$ стало причиной данного $\boldsymbol{v}$. Основной недостаток этого подхода – в том, что так можно обучить сеть вывода только на значениях $\boldsymbol{v}$, имеющих высокую вероятность в модели. На ранних стадиях обучения модельное распределение мало напоминает распределение данных, поэтому сеть вывода не получит возможности обучиться на примерах, похожих на данные.

В разделе 18.2 мы привели возможное объяснение роли сновидений в жизни человека и животных: сны могут поставлять примеры для отрицательной фазы, которыми алгоритмы обучения методами Монте-Карло пользуются для аппроксимации отрицательного градиента логарифма статистической суммы в неориентированных моделях. Другое возможное объяснение биологических сновидений состоит в том, что оно поставляет примеры из распределения $p(\boldsymbol{h}, \boldsymbol{v})$, которые можно использовать,

чтобы обучить сеть вывода предсказанию $\boldsymbol{h}$ по $\boldsymbol{v}$. В некоторых отношениях это объяснение лучше того, в котором фигурирует статистическая сумма. Методы Монте-Карло в общем случае показывают неважные результаты, если в течение нескольких шагов работают только в положительной фазе градиента, а затем в течение нескольких шагов – только в отрицательной фазе. Люди и животные обычно несколько часов подряд бодрствуют, а затем несколько часов спят. Не вполне понятно, как такой график мог бы помочь обучению неориентированной модели методами Монте-Карло. С другой стороны, алгоритмы обучения, основанные на максимизации $\mathcal{L}$, могут в течение длительного периода работать над улучшением q, а затем в течение столь же длительного периода над улучшением $\boldsymbol{\theta}$. Если роль биологических сновидений состоит в том, чтобы обучить сеть предсказанию q, то это объясняет, как животные умудряются несколько часов бодрствовать (чем дольше, тем больше разрыв между $\mathcal{L}$ и $\log p(\boldsymbol{v})$, но $\mathcal{L}$ все время остается нижней границей) и несколько часов спать (во время сна сама порождающая модель не изменяется), не причиняя вреда своим внутренним моделям. Конечно, все это чисто умозрительные заключения, нет никаких доказательств того, что сон преследует именно такие цели. Сновидения могут также служить целям обучения с подкреплением, а не вероятностного моделирования, посредством выполнения выборки синтетического опыта из переходной модели животного, на основе которой обучается стратегия поведения животного. А возможно, что цель сна совершенно иная, и специалисты по машинному обучению еще даже не приблизились к ее пониманию.

19.5.2. Другие формы обученного вывода

Стратегия обученного приближенного вывода применялась и к другим моделям. В работе Salakhutdinov and Larochelle (2010) показано, что один проход обученной сети вывода может дать результаты быстрее, чем итеративное решение уравнений неподвижной точки среднего поля в глубоких машинах Больцмана. Процедура обучения основана на выполнении сети вывода, затем одном шаге среднего поля для улучшения оценок и обучении сети вывода формировать на выходе эту уточненную оценку вместо оригинальной.

В разделе 14.8 мы видели, что модель предсказательной разреженной декомпозиции обучает мелкую сеть кодирования предсказывать разреженный код для входа. Это можно рассматривать как гибрид автокодировщика и разреженного кодирования. Можно придумать вероятностную семантику модели, в рамках которой кодировщик будет выполнять обученный приближенный MAP-вывод. Поскольку кодировщик мелкий, предсказательная разреженная декомпозиция не сможет реализовать ту конкуренцию между блоками, с которой мы встречались при описании вывода среднего поля. Однако с этой проблемой можно справиться, обучив глубокий кодировщик выполнять обученный приближенный вывод, как в методе ISTA (Gregor and LeCun, 2010b).

Обученный приближенный вывод в последнее время стал одним из преобладающих подходов к порождающему моделированию – в форме вариационного автокодировщика (Kingma, 2013; Rezende et al., 2014). В этом элегантном подходе нет нужды явно конструировать цели для сети вывода. Вместо этого сеть вывода просто используется для определения $\mathcal{L}$, а затем параметры этой сети адаптируются для увеличения $\mathcal{L}$. Эта модель подробно описана в разделе 20.10.3.

С помощью приближенного вывода можно обучать и использовать широкий спектр моделей, многие из которых описаны в следующей главе.

Глава 20

Глубокие порождающие модели

В этой главе мы опишем несколько порождающих моделей специального вида, которые можно построить и обучить, применяя методы из глав 16–19. Все эти модели так или иначе представляют распределения вероятности нескольких случайных величин. В одних функцию распределения вероятности можно вычислить явно, в других это невозможно, но зато поддерживаются операции, неявно использующие знания о распределении, например выборка. Одни представляют собой структурные вероятностные модели, описываемые в терминах графов и факторов на языке графических моделей, с которым мы познакомились в главе 16. Другие нельзя описать в терминах факторов, но распределения вероятности они тем не менее представляют.

20.1. Машины Больцмана

Машины Больцмана первоначально были предложены как общий «коннекционистский» подход к обучению произвольных распределений вероятности бинарных векторов (Fahlman et al., 1983; Ackley et al., 1985; Hinton et al., 1984; Hinton and Sejnowski, 1986). Но варианты машин Больцмана со случайными величинами других видов по популярности давно превзошли оригинал. В этом разделе мы кратко рассмотрим бинарную машину Больцмана и обсудим, какие проблемы возникают при попытке обучить модель и выполнить с ее помощью вывод.

Мы определим машину Больцмана над d-мерным бинарным случайным вектором $\mathbf{x} \in \{0, 1\}^d$. Машина Больцмана – это энергетическая модель (см. раздел 16.2.4), т. е. совместное распределение вероятности определяется с помощью функции энергии:

$$P(\boldsymbol{x}) = \frac{\exp(-E(\boldsymbol{x}))}{Z}, \tag{20.1}$$

где $E(\boldsymbol{x})$ – функция энергии, а Z – статистическая сумма, гарантирующая, что $\Sigma_x P(\boldsymbol{x}) = 1$. Функция энергии машины Больцмана имеет вид

$$E(\boldsymbol{x}) = -\boldsymbol{x}^\top \boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x}, \tag{20.2}$$

где $\boldsymbol{U}$ – матрица «весов», содержащая параметры модели, а $\boldsymbol{b}$ – вектор смещений.

Для машины Больцмана общего вида мы имеем набор n-мерных обучающих примеров, а формула (20.1) описывает совместное распределение вероятности наблюдаемых переменных. Ситуация вполне рабочая, но виды взаимодействий между

наблюдаемыми переменными ограничены матрицей весов. Точнее, вероятность, что некоторый блок включен, определяется линейной моделью (логистической регрессией) по значениям других блоков.

Мощность машины Больцмана возрастает, если не все переменные наблюдаемые. В таком случае латентные переменные могут действовать подобно скрытым блокам в многослойном перцептроне и моделировать взаимодействия высшего порядка между видимыми блоками. Напомним, что добавление скрытых блоков в модель логистической регрессии приводит к МСП, который является универсальным аппроксиматором функций. Точно так же машина Больцмана со скрытыми блоками может использоваться уже не только для моделирования линейных связей между переменными, а становится универсальным аппроксиматором функций вероятности для дискретных случайных величин (Le Roux and Bengio, 2008).

Формально говоря, мы разбиваем множество блоков $\boldsymbol{x}$ на два подмножества: видимые $\boldsymbol{v}$ и скрытые $\boldsymbol{h}$. Функция энергии принимает вид

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^\top \boldsymbol{R}\boldsymbol{v} - \boldsymbol{v}^\top \boldsymbol{W}\boldsymbol{h} - \boldsymbol{h}^\top \boldsymbol{S}\boldsymbol{h} - \boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h}. \tag{20.3}$$

Обучение машины Больцмана. Алгоритмы обучения машин Больцмана обычно основаны на критерии максимального правдоподобия. У всех машин Больцмана статистическая сумма вычислительно неразрешима, поэтому градиент максимального правдоподобия следует аппроксимировать методами, описанными в главе 18.

Если правила обучения основаны на максимальном правдоподобии, то у машин Больцмана появляется интересное свойство: обновление веса связи между двумя блоками зависит только от статистик этих двух блоков, собираемых относительно двух разных распределений: $P_{\text{model}}(\boldsymbol{v})$ и $\hat{P}_{\text{data}}(\boldsymbol{v})P_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{v})$. Вся остальная сеть участвует в формировании этих статистик, но для обновления веса не нужно ничего знать ни об остальной сети, ни о том, как собиралась статистика. Следовательно, правило обучения «локально», что придает обучению машины Больцмана некоторое биологическое правдоподобие. Можно предположить, что если бы каждый нейрон был случайной величиной в машине Больцмана, то аксоны и дендриты, соединяющие две случайные величины, могли бы обучаться только путем наблюдения закономерностей возбуждения клеток, с которыми у них имеется физический контакт. В частности, в положительной фазе усиливается связь между двумя блоками, которые часто активируются вместе. Это пример правила обучения Хебба (Hebb, 1949) которое иногда выражают в виде мнемонической фразы: «fire together, wire together» (между нейронами, которые возбуждаются вместе, устанавливается связь). Правила обучения Хебба принадлежат к числу самых старых гипотетических объяснений обучения в биологических системах и сохраняют актуальность по сей день (Giudice et al., 2009).

Другие алгоритмы обучения, в которых используется больше информации, чем просто локальная статистика, похоже, требуют гипотез о наличии дополнительных механизмов. Например, чтобы мозг мог реализовать обратное распространение, как в многослойном перцептроне, кажется необходимым поддержание вторичной коммуникационной сети для передачи информации о градиенте назад по сети. Предложения биологически правдоподобных реализаций (и аппроксимаций) обратного распространения выдвигались (Hinton, 2007a; Bengio, 2015), но пока не проверены, а в работе Bengio (2015) обратное распространение градиентов связывается с выводом в энергетических моделях, подобных машине Больцмана (но с непрерывными латентными переменными).

Отрицательную фазу обучения машины Больцмана объяснить с биологической точки зрения труднее. В разделе 18.2 выдвигалось предположение, что сновидения могут быть формой выборки в отрицательной фазе. Впрочем, эта идея скорее умозрительная.

20.2. Ограниченные машины Больцмана

Ограниченная машина Больцмана, названная изобретателем (Smolensky, 1986) **гармониумом**, сейчас является самым распространенным строительным блоком глубоких вероятностных моделей. Мы вкратце описывали ОМБ в разделе 16.7.1, а здесь напомним сказанное ранее и некоторые дополнительные детали. ОМБ представляет собой неориентированную вероятностную графическую модель, содержащую слой наблюдаемых переменных и единственный слой латентных переменных. ОМБ можно собирать в стек (одна поверх другой) для формирования более глубоких моделей. На рис. 20.1 приведено несколько примеров. Так, на рис. 20.1а изображена графовая структура самой ОМБ. Это двудольный граф, в котором запрещены связи внутри слоя наблюдаемых переменных и внутри слоя латентных переменных.

Мы начнем с бинарной версии ограниченной машины Больцмана, но, как вскоре станет ясно, существуют обобщения на другие типы видимых и скрытых блоков.

Формально говоря, предположим, что наблюдаемый слой состоит из множества n_v бинарных случайных величин, которое будем обозначать вектором $\boldsymbol{v}$. А скрытый слой, состоящий из n_h бинарных случайных величин, обозначим $\boldsymbol{h}$.

Как и общая машина Больцмана, ограниченная машина Больцмана – это энергетическая модель, в которой совместное распределение вероятности описывается функцией энергии:

$$P(\mathbf{v} = \boldsymbol{v}, \mathbf{h} = \boldsymbol{h}) = (1/Z)\exp(-E(\boldsymbol{v}, \boldsymbol{h})). \tag{20.4}$$

Функция энергии ОМБ имеет вид

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h} - \boldsymbol{v}^\top \boldsymbol{W}\boldsymbol{h}, \tag{20.5}$$

а Z – нормировочная постоянная, называемая статистической суммой:

$$Z = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\}. \tag{20.6}$$

Из определения статистической суммы Z ясно, что наивный метод ее вычисления (суммирование по всем состояниям) может оказаться вычислительно неразрешимым, если только не придумать какого-нибудь хитрого алгоритма, который мог бы воспользоваться присутствующей в распределении вероятности регулярностью для более быстрого вычисления. В случае ограниченной машины Больцмана в работе Long and Servedio (2010) формально доказано, что статистическая сумма Z неразрешима. А это означает, что неразрешимым является также совместное распределение $P(\boldsymbol{v})$.

20.2.1. Условные распределения

Хотя $P(\boldsymbol{v})$ неразрешима, у двудольного графа, описывающего структуру ОМБ, есть специальное свойство: условные распределения $P(\mathbf{h} \mid \mathbf{v})$ и $P(\mathbf{v} \mid \mathbf{h})$ факторные, допускающие сравнительно простое вычисление и выборку.

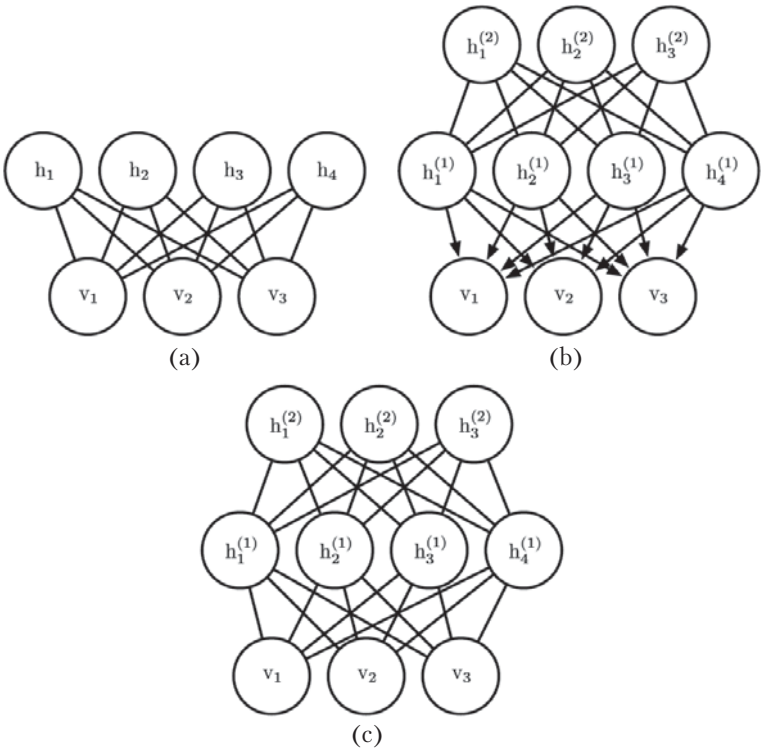

Рис. 20.1 ❖ Примеры моделей, построенных из ограниченных машин Больцмана. (*a*) Сама ограниченная машина Больцмана – это неориентированная графическая модель, основанная на двудольном графе, в одной доле которого находятся видимые блоки, а в другой – скрытые блоки. Между видимыми блоками нет никаких связей – так же, как между скрытыми. Обычно каждый видимый блок связан с каждым скрытым, но встречаются и ОМБ с разреженными связями, например сверточные ОМБ. (*b*) Глубокая сеть доверия (ГСД, англ. DBN)) – гибридная графическая модель, включающая как ориентированные, так и неориентированные связи. Как и в ОМБ, в ней нет внутрислойных связей. Однако в ГСД несколько скрытых слоев, поэтому возможны связи между скрытыми блоками на разных уровнях. Все локальные условные распределения вероятности, необходимые глубокой сети доверия, копируются непосредственно из локальных условных распределений вероятности, составляющих сеть ОМБ. Можно было бы вместо этого представить глубокую сеть доверия полностью неориентированным графом, но тогда потребовались бы внутрислойные связи для улавливания зависимостей между родителями. (*c*) Глубокая машина Больцмана (ГМБ, англ. DBM) – это неориентированная графическая модель с несколькими слоями латентных переменных. У ГМБ, как и у ОМБ и ГСД, нет внутрислойных связей. ГМБ не так тесно связаны с ОМБ, как ГСД. Если ГМБ инициализируется стеком ОМБ, то параметры ОМБ необходимо немного модифицировать. Некоторые виды ГМБ можно обучать без предварительного обучения набора ОМБ

Вывести условные распределения из совместного просто:

$$P(\boldsymbol{h} \mid \boldsymbol{v}) = \frac{P(\boldsymbol{h}, \boldsymbol{v})}{P(\boldsymbol{v})} \tag{20.7}$$

$$= \frac{1}{P(\boldsymbol{v})} \frac{1}{Z} \exp\{\boldsymbol{b}^\top \boldsymbol{v} + \boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}\} \tag{20.8}$$

$$= \frac{1}{Z'} \exp\{\boldsymbol{c}^\top \boldsymbol{h} + \boldsymbol{v}^\top \boldsymbol{W} \boldsymbol{h}\} \tag{20.9}$$

$$= \frac{1}{Z'} \exp\left\{ \sum_{j=1}^{n_h} c_j h_j + \sum_{j=1}^{n_h} \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \boldsymbol{h}_j \right\} \tag{20.10}$$

$$= \frac{1}{Z'} \prod_{j=1}^{n_h} \exp\{c_j h_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j} \boldsymbol{h}_j\}. \tag{20.11}$$

Поскольку в условии находятся видимые блоки **v**, мы можем рассматривать их как постоянные относительно распределения $P(\mathbf{h} \mid \mathbf{v})$. Факторная природа условного распределения $P(\mathbf{h} \mid \mathbf{v})$ сразу же следует из возможности записать совместное распределение вектора $\boldsymbol{h}$ в виде произведения (ненормированных) распределений отдельных элементов h_j. Осталось только нормировать распределения индивидуальных бинарных h_j.

$$P(h_j = 1 \mid \boldsymbol{v}) = \frac{\tilde{P}(h_j = 1 \mid \boldsymbol{v})}{\tilde{P}(h_j = 0 \mid \boldsymbol{v}) + \tilde{P}(h_j = 1 \mid \boldsymbol{v})} \tag{20.12}$$

$$= \frac{\exp\{c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j}\}}{\exp\{0\} + \exp\{c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j}\}} \tag{20.13}$$

$$= \sigma(c_j + \boldsymbol{v}^\top \boldsymbol{W}_{:,j}). \tag{20.14}$$

Теперь можно выразить полное условное распределение скрытого слоя в виде факторного распределения:

$$P(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_{j=1}^{n_h} \sigma((2\boldsymbol{h} - 1) \odot (\boldsymbol{c} + \boldsymbol{W}^\top \boldsymbol{v}))_j. \tag{20.15}$$

Точно так же можно показать, что и другое условное распределение, $P(\boldsymbol{v} \mid \boldsymbol{h})$, является факторным:

$$P(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_{i=1}^{n_v} \sigma((2\boldsymbol{v} - 1) \odot (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}))_i. \tag{20.16}$$

20.2.2. Обучение ограниченных машин Больцмана

Поскольку ОМБ допускает эффективное вычисление и дифференцирование $\tilde{P}(\boldsymbol{v})$, а также эффективную МСМС-выборку в виде блочной выборки по Гиббсу, то ее легко обучить любым из описанных в главе 18 методов обучения моделей с неразрешимыми статистическими суммами: CD, SML (PCD), сопоставление отношений и т. д. По сравнению с другими неориентированными моделями, используемыми в глубоком обучении, ОМБ обучается относительно просто, поскольку мы можем точно вычислить $P(\mathbf{h} \mid \boldsymbol{v})$ в замкнутой форме. В других глубоких моделях, в частности в глубо-

кой машине Больцмана, проблема неразрешимой статистической суммы сочетается с проблемой неразрешимого вывода.

20.3. Глубокие сети доверия

Глубокие сети доверия (ГСД) были одними из первых несверточных моделей, которые удалось успешно обучить в контексте глубоких архитектур (Hinton et al., 2006; Hinton, 2007b). С появлением глубоких сетей доверий в 2006 году началось возрождение глубокого обучения, продолжающееся и по сей день. До этого считалось, что глубокие модели с трудом поддаются оптимизации. Усилия исследователей были в основном направлены на изучение ядерных методов с выпуклыми целевыми функциями. Глубокие сети доверия продемонстрировали, что глубокая архитектура может по качеству превзойти метод опорных векторов с ядром на наборе MNIST (Hinton et al., 2006). Сегодня глубокие сети доверия вышли из моды и применяются редко даже по сравнению с другими порождающими или обучаемыми без учителя алгоритмами, но их роль в истории глубокого обучения достойна всяческого уважения.

Глубокие сети доверия – это порождающие модели с несколькими слоями латентных переменных. Латентные переменные обычно бинарные, хотя видимые блоки могут быть как бинарными, так и вещественными. Внутри слоев нет никаких связей. Обычно каждый блок любого слоя связан с каждым блоком соседних слоев, хотя можно строить и ГСД с более разреженными связями. Связи между двумя верхними слоями неориентированные. Связи между всеми остальными слоями ориентированные, причем стрелки направлены в сторону слоя, ближайшего к данным. Пример показан на рис. 20.1b.

ГСД с l скрытыми слоями содержит l матриц весов $\boldsymbol{W}^{(1)}$, ..., $\boldsymbol{W}^{(l)}$, а также $l + 1$ векторов смещений $\boldsymbol{b}^{(0)}$, ..., $\boldsymbol{b}^{(l)}$, где $\boldsymbol{b}^{(0)}$ – смещения для видимого слоя. Распределение вероятности, представляемое ГСД, имеет вид:

$$P(\boldsymbol{h}^{(l)}, \boldsymbol{h}^{(l-1)}) \propto \exp(\boldsymbol{b}^{(l)\top}\boldsymbol{h}^{(l)} + \boldsymbol{b}^{(l-1)\top}\boldsymbol{h}^{(l-1)} + \boldsymbol{h}^{(l-1)\top}\boldsymbol{W}^{(l)}\boldsymbol{h}^{(l)}), \tag{20.17}$$

$$P(h_i^{(k)} = 1 \mid \boldsymbol{h}^{(k+1)}) = \sigma(b_i^{(k)} + \boldsymbol{W}_{:,i}^{(k+1)\top}\boldsymbol{h}^{(k+1)}) \forall i, \forall k \in 1, ..., l-2, \tag{20.18}$$

$$P(v_i = 1 \mid \boldsymbol{h}^{(1)}) = \sigma(b_i^{(0)} + \boldsymbol{W}_{:,i}^{(1)\top}\boldsymbol{h}^{(1)}) \forall i. \tag{20.19}$$

В случае вещественных видимых блоков подставляем

$$\mathbf{v} \sim \mathcal{N}(\boldsymbol{v}; \boldsymbol{b}^{(0)} + \boldsymbol{W}^{(1)\top}\boldsymbol{h}^{(1)}, \boldsymbol{\beta}^{-1}), \tag{20.20}$$

где $\boldsymbol{\beta}$ – диагональная матрица, иначе вычисления становятся слишком сложными. Обобщение на другие экспоненциальные семейства видимых блоков не вызывают трудностей, по крайней мере в теории. ГСД с единственным скрытым слоем – это просто ОМБ.

Чтобы произвести выборку из ГСД, мы сначала выполняем несколько шагов выборки по Гиббсу для двух верхних скрытых слоев. На этом этапе, по существу, производится выборка из ОМБ, определенной этими двумя слоями. Затем можно применить один проход предковой выборки к остальной части модели, чтобы произвести выборку из видимых блоков.

Глубокие сети доверия подвержены многим проблемам, присущим как ориентированным, так и неориентированным моделям.

Вывод в глубокой сети доверия вычислительно неразрешим из-за эффекта оправдания внутри каждого ориентированного слоя и взаимодействия между двумя скры-

тыми слоями с неориентированными связями. Вычисление или максимизация стандартной нижней границы свидетельств для логарифмического правдоподобия также неразрешимы, поскольку в этой нижней границе участвует математическое ожидание клик, размер которых равен ширине сети.

При вычислении или максимизации логарифмического правдоподобия приходится сталкиваться не только с проблемой неразрешимого вывода для исключения латентных переменных, но и с проблемой неразрешимой статистической суммы в неориентированной модели двух верхних слоев.

Обучение глубокой сети доверия начинается с того, что мы обучаем ОМБ максимизировать $\mathbb{E}_{\mathbf{v}\sim p_{\text{data}}} \log p(\boldsymbol{v})$ с помощью алгоритма сопоставительного расхождения или стохастической максимизации правдоподобия. Полученные параметры ОМБ определяют параметры первого слоя ГСД. Далее мы обучаем вторую ЛМБ приближенно максимизировать выражение

$$\mathbb{E}_{\mathbf{v}\sim p_{\text{data}}} \mathbb{E}_{\mathbf{h}^{(1)}\sim p^{(1)}(\boldsymbol{h}^{(1)}|\boldsymbol{v})} \log p^{(2)}(\boldsymbol{h}^{(1)}), \tag{20.21}$$

где $p^{(1)}$ – распределение вероятности, представленное первой ОМБ, а $p^{(2)}$ – распределение, представленное второй ОМБ. Иными словами, вторая ОМБ обучается моделировать распределение, определенное выборкой из скрытых блоков первой ОМБ, тогда как первая ОМБ управляется данными. Эту процедуру можно повторять бесконечно, добавляя в ГСД столько слоев, сколько необходимо, при этом каждая новая ОМБ будет моделировать выборку из предыдущей. Каждая ОМБ определяет очередной слой ГСД. Эту процедуру можно обосновать как увеличение вариационной нижней границы логарифмического правдоподобия данных в модели ГСД (Hinton et al., 2006).

В большинстве приложений не предпринимается никаких усилий для совместного обучения ГСД после завершения жадной послойной процедуры. Однако можно провести окончательную настройку с помощью алгоритма бодрствования-сна.

Обученную ГСД можно использовать непосредственно в качестве порождающей модели, но интерес к ГСД связан в основном с их способностью улучшать модели классификации. Мы можем воспользоваться весами из ГСД для определения многослойного перцептрона:

$$\boldsymbol{h}^{(1)} = \sigma(b^{(1)} + \boldsymbol{v}^{\top}\boldsymbol{W}^{(1)}), \tag{20.22}$$

$$\boldsymbol{h}^{(l)} = \sigma(b_i^{(l)} + \boldsymbol{h}^{(l-1)\top}\boldsymbol{W}^{(l)})\forall l \in 2, ..., m. \tag{20.23}$$

После инициализации МСП весами и смещениями, обученными на этапе порождающего обучения ГСД, мы можем обучить МСП решать задачу классификации. Это дополнительное обучение – пример дискриминантной окончательной настройки.

Этот конкретный выбор МСП выглядит несколько произвольно, по сравнению со многими уравнениями вывода из главы 19, имеющими теоретическое обоснование. Это эвристический выбор, который неплохо работает на практике и постоянно упоминается в литературе. В обоснование многих методов приближенного вывода выдвигается их способность находить максимально *точную* вариационную нижнюю границу логарифмического правдоподобия при определенных ограничениях. Такую границу можно построить с помощью математических ожиданий скрытых блоков, определенных ассоциированным с ГСД перцептроном, но это справедливо для *любого* распределения вероятности скрытых блоков, и нет причин полагать, что этот МСП

дает особо точную границу. В частности, МСП игнорирует многие важные взаимодействия в графической модели ГСД. МСП распространяет информацию вверх от видимых блоков к самым глубоким скрытым блокам, но не вниз и не в стороны. В графической модели ГСД имеются оправдывающие взаимодействия между всеми скрытыми блоками внутри одного слоя, а также нисходящие взаимодействия между слоями. Хотя логарифмическое правдоподобие ГСД вычислить невозможно, его можно аппроксимировать методом выборки по значимости с отжигом (Salakhutdinov and Murray, 2008). Это позволяет вычислить ее качество в роли порождающей модели.

Термин «глубокая сеть доверия» часто неправильно употребляется для обозначения любой глубокой нейронной сети, даже без семантики латентных переменных. На самом деле он относится исключительно к моделям с неориентированными связями в самом глубоком слое и направленными вниз ориентированными связями между всеми остальными парами соседних слоев.

Этот термин также может вызвать путаницу, потому что «сетью доверия» иногда называют строго ориентированные модели, тогда как в глубоких сетях доверия имеется неориентированный слой. Ко всему прочему, английский акроним DBN употребляется также для динамических байесовских сетей (Dean and Kanazawa, 1989), т. е. для байесовских сетей, представляющих марковские цепи.

20.4. Глубокие машины Больцмана

Глубокая машина Больцмана (ГМБ) (Salakhutdinov and Hinton, 2009a) – еще один вид порождающих моделей. В отличие от глубокой сети доверия, она полностью неориентированная, а в отличие от ОМБ, имеет несколько слоев латентных переменных (в ОМБ такой слой единственный). Но так же, как и в ОМБ, внутри слоя все переменные взаимно независимы при условии переменных из соседних слоев. Структура графа показана на рис. 20.2. Глубокие машины Больцмана применялись к различным задачам, в т. ч. к моделированию документов (Srivastava et al., 2013).

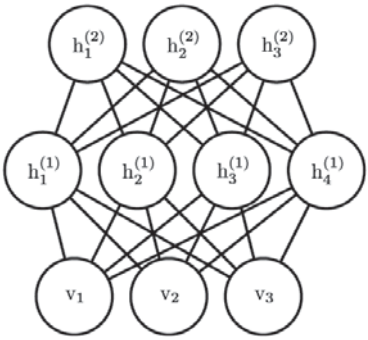

Рис. 20.2 ❖ Графическая модель глубокой машины Больцмана с одним видимым слоем (внизу) и двумя скрытыми слоями. Связи существуют только между блоками из соседних слоев. Внутри слоев никаких связей нет

Подобно ОМБ и ГСД, глубокие машины Больцмана обычно содержат только бинарные блоки (и мы придерживаемся такого предположения для простоты изложения), но включить в них вещественные видимые блоки не составляет труда.

ОМБ – энергетическая модель, а значит, совместное распределение вероятности переменных модели параметризовано функцией энергии E. В случае глубокой машины Больцмана с одним видимым слоем $\boldsymbol{v}$ и тремя скрытыми слоями $\boldsymbol{h}^{(1)}$, $\boldsymbol{h}^{(2)}$, $\boldsymbol{h}^{(3)}$ совместное распределение имеет вид:

$$P(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(-E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}; \boldsymbol{\theta})). \tag{20.24}$$

Чтобы упростить изложение, мы далее опускаем параметры смещения. Тогда функция ГМБ определяется формулой:

$$E(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \boldsymbol{h}^{(3)}; \boldsymbol{\theta}) = -\boldsymbol{v}^{\top} \boldsymbol{W}^{(1)} \boldsymbol{h}^{(1)} - \boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)} \boldsymbol{h}^{(2)} - \boldsymbol{h}^{(2)\top} \boldsymbol{W}^{(3)} \boldsymbol{h}^{(3)}. \tag{20.25}$$

По сравнению с функцией энергии ОМБ (20.5), функция энергии ГМБ включает связи между скрытыми блоками (латентными переменными) в форме матриц весов ($\boldsymbol{W}^{(2)}$ и $\boldsymbol{W}^{(3)}$). Как мы увидим, наличие этих связей имеет важные последствия для поведения модели, а также для выполнения вывода.

По сравнению с полносвязными машинами Больцмана (в которых каждый блок связан со всеми остальными), ГМБ имеет ряд преимуществ, похожих на те, что свойственны ОМБ. Точнее говоря, как видно по рис. 20.3, слои ГМБ можно представить в виде двудольного графа, в котором нечетные слои принадлежат одной доле, а четные – другой. Отсюда сразу следует, что при условии переменных в четном слое переменные в нечетных слоях становятся условно независимыми. Разумеется, обусловливание переменными нечетных слоев делает условно независимыми переменные в четных слоях.

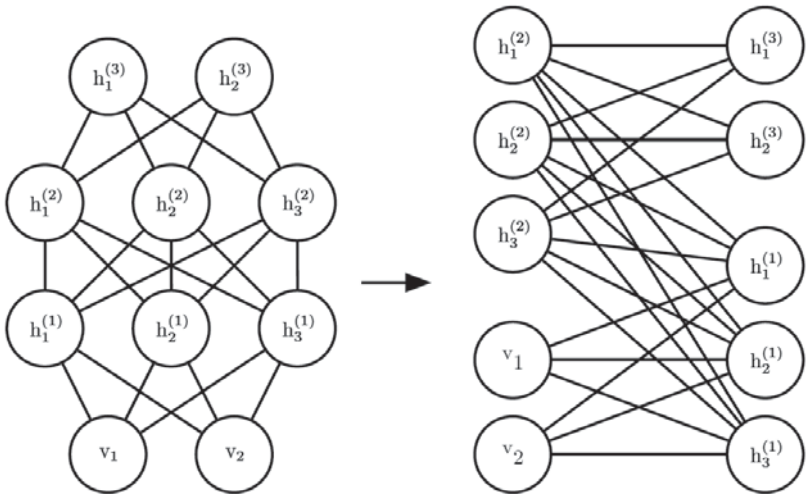

Рис. 20.3 ❖ Глубокая машина Больцмана, нарисованная так, чтобы выявить структуру двудольного графа

Из двудольной структуры ГМБ вытекает, что те же самые уравнения, которые мы раньше использовали для условных распределений ОМБ, применимы и к ГМБ. Блоки внутри одного слоя условно независимы друг от друга при условии значений в соседних слоях, поэтому распределения бинарных переменных можно полностью

описать параметрами Бернулли, задающими вероятность активности каждого блока. В нашем примере с двумя скрытыми слоями вероятности активации равны

$$P(v_i = 1 \mid \boldsymbol{h}^{(1)}) = \sigma(\boldsymbol{W}^{(1)}_{i,:}\boldsymbol{h}^{(1)}), \tag{20.26}$$

$$P(h^{(1)}_i = 1 \mid \boldsymbol{v}, \boldsymbol{h}^{(2)}) = \sigma(\boldsymbol{v}^\top \boldsymbol{W}^{(1)}_{:,i} + \boldsymbol{W}^{(2)}_{i,:}\boldsymbol{h}^{(2)}), \tag{20.27}$$

и

$$P(h^{(2)}_k = 1 \mid \boldsymbol{h}^{(1)}) = \sigma(\boldsymbol{h}^{(1)\top} \boldsymbol{W}^{(2)}_{:,k}). \tag{20.28}$$

Благодаря двудольной структуре глубокой машины Больцмана оказывается эффективной выборка по Гиббсу. При наивном подходе к выборке по Гиббсу обновляется по одной переменной за раз. ОМБ позволяет обновлять все видимые блоки в одной операции, а все скрытые – в другой. По наивности можно было бы предположить, что ГМБ с l слоями требует $l + 1$ обновлений, так что на каждой итерации обновляются все блоки одного уровня. На самом же деле для обновления всех блоков достаточно всего двух итераций. Выборку по Гиббсу можно разбить на две группы обновлений: одна включает все четные слои (в т. ч. и видимый), другая – все нечетные. В силу двудольности графа связей в ГМБ распределение нечетных слоев при условии четных факторное, поэтому выборку из него можно произвести одновременно и независимо одной операцией. И так же можно произвести одновременную и независимую выборку из распределения четных слоев при условии нечетных. Эффективная выборка особенно важна для обучения с помощью алгоритма стохастической максимизации правдоподобия.

20.4.1. Интересные свойства

У глубоких машин Больцмана много интересных свойств.

ГМБ были разработаны после ГСД. По сравнению с ГСД, их апостериорное распределение $P(\boldsymbol{h} \mid \boldsymbol{v})$ проще. В противоречие с интуицией, благодаря простоте апостериорного распределения возможны его улучшенные аппроксимации. В случае ГСД мы выполняем классификацию, применяя эвристическую процедуру приближенного вывода, в которой высказываем гипотезу, что разумное значение математического ожидания среднего поля для скрытых блоков можно получить проходом вверх по сети, образованной МСП с сигмоидными функциями активации и такими же весами, как у исходной ГСД. Для получения вариационной нижней границы логарифмического правдоподобия можно использовать любое распределение $Q(\boldsymbol{h})$. Поэтому такая эвристическая процедура дает возможность получить нижнюю границу. Однако эта граница явно никак не оптимизировалась и потому может быть далеко не точной. В частности, эвристическая оценка Q игнорирует взаимодействия между скрытыми блоками в одном слое, а также влияние нисходящей обратной связи между скрытыми блоками более глубоких слоев и слоев, расположенных ближе к входу. Поскольку эвристическая процедура вывода на основе МСП не может учесть этих взаимодействий в ГМБ, результирующее распределение Q, скорее всего, далеко от оптимального. В ГМБ все скрытые блоки внутри одного слоя условно независимы при условии других слоев. Благодаря отсутствию внутрислойного взаимодействия открывается возможность использовать уравнения неподвижной точки для оптимизации вариационной нижней границы и нахождения истинно оптимальных математических ожиданий среднего поля (в пределах некоторого допуска).

Использование надлежащего среднего поля позволяет процедуре приближенного вывода в ГМБ уловить влияние нисходящей обратной связи. Это делает ГМБ интересными для нейробиологии, поскольку известно, что человеческий мозг задействует много нисходящих обратных связей. Благодаря этому свойству ГМБ использовались в качестве вычислительных моделей реальных нейробиологических явлений (Series et al., 2010; Reichert et al., 2011).

Один из недостатков ГМБ – относительная сложность выборки из них. В ГСД выборку МСМС-методами необходимо использовать только в двух верхних слоях. Остальные слои используются лишь в конце процесса выборки, в одном эффективном проходе предковой выборки. Чтобы произвести выборку из ГМБ, необходимо применять МСМС-методы во всех слоях, т. е. каждый слой модели принимает участие во всех переходах марковской цепи.

20.4.2. Вывод среднего поля в ГМБ

Условное распределение одного слоя ГМБ при условии соседних слоев факторное. В примере ГМБ с двумя скрытыми слоями это будут распределения $P(\boldsymbol{v} \mid \boldsymbol{h}^{(1)})$, $P(\boldsymbol{h}^{(1)} \mid \boldsymbol{v}, \boldsymbol{h}^{(2)})$ и $P(\boldsymbol{h}^{(2)} \mid \boldsymbol{h}^{(1)})$. Распределение всех скрытых слоев обычно не является факторным из-за взаимодействий между слоями. В примере с двумя скрытыми слоями $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$ не факторизуется из-за весов $\boldsymbol{W}^{(2)}$ взаимодействия между $\boldsymbol{h}^{(1)}$ и $\boldsymbol{h}^{(2)}$, вследствие чего эти переменные оказываются взаимно зависимыми.

Как и в случае с ГСД, нам остается искать способы аппроксимации апостериорного распределения ГМБ. Но, в отличие от ГСД, апостериорное распределение скрытых блоков ГМБ, хотя и сложное, легко аппроксимируется вариационной аппроксимацией (см. раздел 19.4), а конкретно – приближением среднего поля. Приближение среднего поля – это простая форма вариационного вывода, когда мы ограничиваемся только факторными аппроксимирующими распределениями. В контексте ГМБ уравнения среднего поля улавливают двусторонние взаимодействия между слоями. В этом разделе мы построим итеративную процедуру приближенного вывода, впервые предложенную в работе Salakhutdinov and Hinton (2009a).

Вариационный подход к приближенному выводу предполагает аппроксимацию конкретного целевого распределения – в нашем случае апостериорного распределения скрытых блоков при условии видимых блоков – некоторым достаточно простым семейством распределений. В случае приближения среднего поля в качестве такого семейства берется множество распределений, для которых скрытые блоки условно независимы.

Теперь разработаем подход на основе среднего поля для примера с двумя скрытыми слоями. Пусть $Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$ – аппроксимация $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. Из предположения среднего поля следует, что

$$Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) = \prod_j Q(h_j^{(1)} \mid \boldsymbol{v}) \prod_k Q(h_k^{(2)} \mid \boldsymbol{v}). \tag{20.29}$$

Приближение среднего поля пытается найти член этого семейства распределений, который наилучшим образом аппроксимирует истинное апостериорное распределение $P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. Важно отметить, что процесс вывода следует запускать снова для нахождения другого распределения Q всякий раз, как используется новое значение $\boldsymbol{v}$.

Можно придумать много способов измерить качество аппроксимации $P(\boldsymbol{h} \mid \boldsymbol{v})$ распределением $Q(\boldsymbol{h} \mid \boldsymbol{v})$. Подход на основе среднего поля предполагает минимизацию

$$\mathrm{KL}(Q \| P) = \sum_{\boldsymbol{h}} Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) \log\left(\frac{Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})}{P(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v})}\right). \tag{20.30}$$

Вообще говоря, мы не обязаны предоставлять параметрическую форму аппроксимирующего распределения, а только гарантировать выполнение предположений о независимости. Процедура вариационной аппроксимации сама способна восстановить функциональную форму приближенного распределения. Однако в рассматриваемом случае предположения скрытого поля о бинарных скрытых блоках фиксирование параметризации модели заранее не приводит к потере общности.

Мы параметризуем Q в виде произведения распределений Бернулли, т. е. ассоциируем параметр с вероятностью каждого элемента $\boldsymbol{h}^{(1)}$. Точнее, для каждого j $\hat{h}_j^{(1)} = Q(h_j^{(1)} = 1 \mid \boldsymbol{v})$, где $\hat{h}_j^{(1)} \in [0, 1]$, и для каждого k $\hat{h}_k^{(2)} = Q(h_k^{(2)} = 1 \mid \boldsymbol{v})$, где $\hat{h}_k^{(2)} \in [0, 1]$. Таким образом, имеем следующую аппроксимацию апостериорного распределения:

$$Q(\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)} \mid \boldsymbol{v}) = \prod_j Q(h_j^{(1)} \mid \boldsymbol{v}) \prod_k Q(h_k^{(2)} \mid \boldsymbol{v}) \tag{20.31}$$

$$= \prod_j (\hat{h}_j^{(1)})^{h_j^{(1)}} (1 - \hat{h}_j^{(1)})^{(1-h_j^{(1)})} \times \prod_k (\hat{h}_k^{(2)})^{h_k^{(2)}} (1 - \hat{h}_k^{(2)})^{(1-h_k^{(2)})}. \tag{20.32}$$

Разумеется, эта параметризация приближенного апостериорного распределения очевидным образом обобщается на ГМБ с большим числом слоев, нужно только воспользоваться двудольной структурой графа и одновременно обновить сначала все четные слои, а затем все нечетные, применяя такую же схему, как в выборке по Гиббсу.

После того как семейство аппроксимирующих распределений Q определено, остается задать процедуру выбора того члена этого семейства, который лучше всего соответствует P. Самое простое – воспользоваться уравнениями среднего поля (19.56). При выводе этих уравнений мы искали, в каких точках обращаются в нуль производные вариационной нижней границы. Они абстрактно описывают, как оптимизировать вариационную нижнюю границу для любой модели, просто взяв математические ожидания относительно Q.

Применив эти общие уравнения, получим правила обновления (опять-таки члены смещения игнорируются):

$$\hat{h}_j^{(1)} = \sigma\left(\sum_i v_i W_{i,j}^{(1)} + \sum_{k'} W_{j,k'}^{(2)} \hat{h}_{k'}^{(2)}\right), \ \forall j, \tag{20.33}$$

$$\hat{h}_k^{(2)} = \sigma\left(\sum_{j'} W_{j',k}^{(2)} \hat{h}_{j'}^{(1)}\right), \ \forall k. \tag{20.34}$$

В неподвижной точке этой системы уравнений мы имеем локальный максимум вариационной нижней границы $\mathcal{L}(Q)$. Следовательно, эти уравнения обновления неподвижной точки определяют итеративный алгоритм, в котором обновление $\hat{h}_j^{(1)}$ (по формуле 20.33) чередуется с обновлением $\hat{h}_k^{(2)}$ (по формуле 20.34). В небольших задачах типа MNIST достаточно всего десяти итераций, чтобы найти приближенный градиент положительной фазы для обучения, а пятидесяти обычно хватает для получения высококачественного представления одного конкретного примера, используемого для классификации с высокой верностью. Обобщение приближенного вариационного вывода на более глубокие ГМБ не составляет труда.

20.4.3. Обучение параметров ГМБ

При обучении ГМБ приходится решать проблему неразрешимой статистической суммы и проблему неразрешимого апостериорного распределения. Для этого применяются соответственно методы из глав 18 и 19.

Как сказано в разделе 20.4.2, вариационный вывод допускает построение распределения $Q(\boldsymbol{h} \mid \boldsymbol{v})$, аппроксимирующего неразрешимое распределение $P(\boldsymbol{h} \mid \boldsymbol{v})$. Затем максимизируется $\mathcal{L}(\boldsymbol{v}, Q, \boldsymbol{\theta})$ – вариационная нижняя граница неразрешимого логарифмического правдоподобия, $\log P(\boldsymbol{v}; \boldsymbol{\theta})$.

Для глубокой машины Больцмана с двумя скрытыми слоями функция $\mathcal{L}$ имеет вид:

$$\mathcal{L}(Q, \boldsymbol{\theta}) = \sum_i \sum_{j'} v_i W^{(1)}_{i,j'} \hat{h}^{(1)}_{j'} + \sum_{j'} \sum_{k'} \hat{h}^{(1)}_{j'} W^{(2)}_{j',k'} \hat{h}^{(2)}_{k'} - \log Z(\boldsymbol{\theta}) + \mathcal{H}(Q). \tag{20.35}$$

Это выражение все еще содержит логарифм статистической суммы $\log Z(\boldsymbol{\theta})$. Поскольку глубокая машина Больцмана состоит из ограниченных машин Больцмана, то результаты, касающиеся трудности вычисления статистической суммы и выборки в ограниченных машинах Больцмана, применимы и к ГМБ. Это означает, что для вычисления функции вероятности машины Больцмана необходимы приближенные методы, например выборка по значимости с отжигом. Аналогично для обучения модели требуется аппроксимировать градиент логарифма статистической суммы. Общее описание этих методов см. в главе 18. ГМБ обычно обучаются с помощью алгоритма стохастической максимизации правдоподобия. Многие другие методы, упомянутые в главе 18, к ГМБ неприменимы. Для псевдоправдоподобия необходимо уметь вычислять ненормированные вероятности, а не просто получать для них вариационную нижнюю границу. Метод сопоставительного расхождения слишком медленный для глубоких машин Больцмана, потому что они не допускают эффективной выборки из распределения скрытых блоков при условии видимых, поэтому метод сопоставительного расхождения должен был бы прирабатывать марковскую цепь всякий раз, как необходим новый пример в отрицательной фазе.

В разделе 18.2 обсуждается невариационная версия алгоритма стохастической максимизации правдоподобия. Порядок применения вариационной стохастической максимизации правдоподобия описан в алгоритме 20.1. Напомним, что мы говорим об упрощенном варианте ГМБ без параметров смещения; впрочем, включить их в рассмотрение очень просто.

20.4.4. Послойное предобучение

К сожалению, обучение ГМБ методом стохастической максимизации правдоподобия (как описано выше) со случайными начальными параметрами не годится. В одних случаях модель не может обучиться адекватному представлению распределения, в других ГМБ представляет распределение хорошо, но не удается получить более высокое правдоподобие, чем дала бы простая ОМБ. ГМБ, для которой веса очень малы во всех слоях, кроме первого, представляет приблизительно то же распределение, что и ОМБ.

В разделе 20.4.5 описаны вариационные методы, допускающие совместное обучение. Однако оригинальный и самый популярный метод решения проблемы совместного обучения ГМБ – жадное послойное предобучение. В этом случае каждый слой

ГМБ обучается изолированно – как ОМБ. Первый слой обучается моделированию входных данных, а каждый последующий – моделированию примеров, выбранных из апостериорного распределения предыдущей ОМБ. После того как все ОМБ обучены, их можно объединить в ГМБ. Затем ГМБ можно обучить методом PCD. Обычно такое обучение вносит лишь небольшое изменение в параметры модели и в ее качество, измеряемое по логарифмическому правдоподобию, присвоенному данным, или по способности модели классифицировать входы. Процедура обучения иллюстрируется на рис. 20.4.

Алгоритм 20.1. Алгоритм вариационной стохастической максимизации правдоподобия для обучения ГМБ с двумя скрытыми слоями

Установить размер шага ε равным малому положительному числу.
Установить число шагов выборки по Гиббсу k достаточно большим для приработки марковской цепи $p(\boldsymbol{v}, \boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}; \boldsymbol{\theta} + \varepsilon\Delta_{\boldsymbol{\theta}})$.

Инициализировать три матрицы $\tilde{\boldsymbol{V}}$, $\tilde{\boldsymbol{H}}^{(1)}$ и $\tilde{\boldsymbol{H}}^{(2)}$ с m строками каждая случайными значениями (например, выбранными из распределений Бернулли с такими же маргиналами, как у модели).
while не сошелся (цикл обучения) **do**
  Выбрать мини-пакет m примеров из обучающих данных и организовать его в виде строк матрицы плана $\boldsymbol{V}$.
  Инициализировать матрицы $\hat{\boldsymbol{H}}^{(1)}$ и $\hat{\boldsymbol{H}}^{(2)}$, возможно, маргиналами модели.
  while не сошелся (цикл вывода среднего поля) **do**
    $\hat{\boldsymbol{H}}^{(1)} \leftarrow \sigma(\boldsymbol{V}\boldsymbol{W}^{(1)} + \hat{\boldsymbol{H}}^{(2)}\boldsymbol{W}^{(2)\top})$
    $\hat{\boldsymbol{H}}^{(2)} \leftarrow \sigma(\hat{\boldsymbol{H}}^{(1)}\boldsymbol{W}^{(2)})$
  end while
  $\Delta_{\boldsymbol{W}^{(1)}} \leftarrow (1/m)\boldsymbol{V}^{\top}\hat{\boldsymbol{H}}^{(1)}$
  $\Delta_{\boldsymbol{W}^{(2)}} \leftarrow (1/m)\hat{\boldsymbol{H}}^{(1)\top}\hat{\boldsymbol{H}}^{(2)}$
  for $l = 1$ to k (выборка по Гиббсу) **do**
    Блочная выборка по Гиббсу 1:
    $\forall i, j, \tilde{V}_{i,j}$ выбирается из $P(\tilde{V}_{i,j} = 1) = \sigma(\boldsymbol{W}^{(1)}_{j,:}(\tilde{\boldsymbol{H}}^{(1)}_{i,:})^{\top})$.
    $\forall i, j, \tilde{H}^{(2)}_{i,j}$ выбирается из $P(\tilde{H}^{(2)}_{i,j} = 1) = \sigma(\tilde{\boldsymbol{H}}^{(1)}_{i,:}\boldsymbol{W}^{(2)}_{:,j})$.
    Блочная выборка по Гиббсу 2:
    $\forall i, j, \tilde{H}^{(1)}_{i,j}$ выбирается из $P(\tilde{H}^{(1)}_{i,j} = 1) = \sigma(\tilde{\boldsymbol{V}}_{i,:}\boldsymbol{W}^{(1)}_{:,j} + \tilde{\boldsymbol{H}}^{(2)}_{i,:}\boldsymbol{W}^{(2)\top}_{j,:})$.
  end for
  $\Delta_{\boldsymbol{W}^{(1)}} \leftarrow \Delta_{\boldsymbol{W}^{(1)}} - (1/m)\boldsymbol{V}^{\top}\tilde{\boldsymbol{H}}^{(1)}$
  $\Delta_{\boldsymbol{W}^{(2)}} \leftarrow \Delta_{\boldsymbol{W}^{(2)}} - (1/m)\tilde{\boldsymbol{H}}^{(1)\top}\tilde{\boldsymbol{H}}^{(2)}$
  $\boldsymbol{W}^{(1)} \leftarrow \boldsymbol{W}^{(1)} + \varepsilon\Delta_{\boldsymbol{W}^{(1)}}$ (это упрощенная иллюстрация, на практике применяется более эффективный алгоритм, например импульсный с убывающей скоростью обучения)
  $\boldsymbol{W}^{(2)} \leftarrow \boldsymbol{W}^{(2)} + \varepsilon\Delta_{\boldsymbol{W}^{(2)}}$
end while

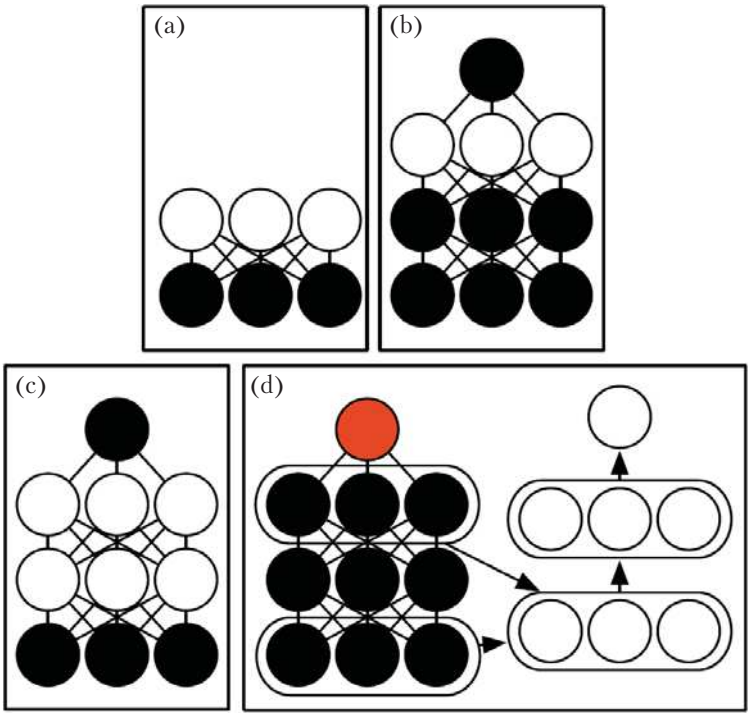

Рис. 20.4 ❖ Процедура обучения глубокой машины Больцмана, использованной для классификации набора данных MNIST (Salakhutdinov and Hinton, 2009a; Srivastava et al., 2014). (*a*) Обучить ОМБ, применив алгоритм CD для приближенной максимизации $\log P(\boldsymbol{v})$. (*b*) Обучить вторую ОМБ, которая моделирует $\boldsymbol{h}^{(1)}$ и целевой класс y, применив алгоритм CD-k для приближенной максимизации $\log P(\boldsymbol{h}^{(1)}, \mathrm{y})$, где $\boldsymbol{h}^{(1)}$ – выборка из апостериорного распределения первой ОМБ при условии данных. Увеличивать k от 1 до 20 в процессе обучения. (*c*) Объединить обе ОМБ в ГМБ. Обучить ее приближенной максимизации $\log P(\mathbf{v}, \mathrm{y})$, применив алгоритм стохастической максимизации правдоподобия с $k = 5$. (*d*) Удалить y из модели. Определить новый набор признаков $\boldsymbol{h}^{(1)}$ и $\boldsymbol{h}^{(2)}$, полученных путем выполнения вывода среднего поля в модели без y. Использовать эти признаки в качестве входа МСП, структура которого такая же, как структура дополнительного прохода среднего поля, с дополнительным выходным слоем для оценки y. Инициализировать веса МСП весами ГМБ. Обучить МСП приближенной максимизации $\log P(\mathrm{y} \mid \mathbf{v})$, применив алгоритм стохастического градиентного спуска и прореживание. Рисунок взят из работы Goodfellow et al. (2013b)

Эта процедура жадного послойного обучения – не просто покоординатное восхождение. Она действительно напоминает покоординатное восхождение, потому что на каждом шаге мы оптимизируем одно подмножество параметров. Но оба метода отличаются, поскольку в процедуре жадного послойного обучения на каждом шаге используется другая целевая функция.

Жадное послойное предобучение ГМБ отличается от жадного послойного предобучения ГСД. Параметры каждой отдельной ОМБ можно копировать в соответствующую ГСД непосредственно. В случае же ГМБ параметры ОМБ необходимо

модифицировать перед включением в ГМБ. Слой в середине стека ОМБ обучается только на входных данных, поступающих снизу, но после того как стек собран в ГМБ, этому слою данные поступают снизу и сверху. Чтобы учесть этот эффект, в работе Salakhutdinov and Hinton (2009a) предлагается делить пополам веса всех ОМБ, кроме нижней и верхней, перед тем как вставлять их в ГМБ. Кроме того, нижнюю ОМБ следует обучать с использованием двух «копий» каждого видимого блока со связанными, равными между собой весами. Это означает, что на восходящем проходе веса, по сути дела, удваиваются. Аналогично верхнюю ОМБ следует обучать с использованием двух копий верхнего слоя.

Для получения не уступающих лучшим образцам результатов с помощью глубоких машин Больцмана необходимо модифицировать стандартный алгоритм стохастической максимизации правдоподобия, а именно использовать небольшую толику среднего поля в отрицательной фазе шага совместного обучения методом PCD (Salakhutdinov and Hinton, 2009a). Точнее говоря, математическое ожидание градиента энергии следует вычислять относительно распределения среднего поля, в котором все блоки независимы. Параметры этого распределения среднего поля следует получать, выполнив всего одну итерацию уравнений неподвижной точки среднего поля. См. работу Goodfellow et al. (2013b), где приведено сравнение качества центрированных ГМБ с применением частичного среднего поля в отрицательной фазе и без оного.

20.4.5. Совместное обучение глубоких машин Больцмана

Для классической ГМБ требуется жадное предобучение без учителя, а чтобы она хорошо выполняла классификацию, необходим отдельный основанный на МСП классификатор поверх выделенных ей скрытых признаков. У этой схемы есть нежелательные свойства. Трудно следить за качеством в процессе обучения, поскольку мы не можем вычислить свойства полной ГМБ во время обучения первой ОМБ. Поэтому сказать, насколько хорошо выбраны гиперпараметры, можно только, когда процесс обучения зайдет достаточно далеко. Программным реализациям ГМБ нужно много различных компонент: для обучения отдельных ОМБ методом CD, обучения полной ГМБ методом PCD и обучения на основе обратного распространения через МСП. Наконец, МСП, построенные поверх машины Больцмана, теряют многие преимущества ее вероятностной модели, например способность выполнять вывод, когда часть входных значений отсутствует.

Существуют два основных способа решить проблему совместного обучения глубокой машины Больцмана. Первый – **центрированная глубокая машина Больцмана** (Montavon and Muller, 2012), когда модель перепараметризуется так, чтобы гессиан функции стоимости был лучше обусловлен в начале процесса обучения. В результате получается модель, которую можно обучить без этапа жадного послойного предобучения. Эта модель достигает отличного логарифмического правдоподобия на тестовом наборе и порождает примеры высокого качества. К сожалению, она попрежнему не может конкурировать с правильно регуляризированным МСП в роли классификатора. Второй способ – использовать **многопредсказательную глубокую машину Больцмана** (multi-prediction deep Boltzmann machine) (Goodfellow et al., 2013b). В этой модели применяется альтернативный критерий обучения, который позволяет использовать алгоритм обратного распространения, чтобы избежать проблем с МСМС-оценками градиента. К сожалению, новый критерий не приводит к хороше-

му правдоподобию или выборкам, но, по сравнению с МСМС-методами, производит более качественную классификацию и может хорошо рассуждать об отсутствующих входных данных.

Центрирование машины Больцмана проще всего описать, вернувшись к общему взгляду на машину Больцмана как на множество блоков $\boldsymbol{x}$ с матрицей весов $\boldsymbol{U}$ и смещениями $\boldsymbol{b}$. Напомним, что функция энергии имеет вид

$$E(\boldsymbol{x}) = -\boldsymbol{x}^\top \boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x}. \tag{20.36}$$

Применяя различные паттерны разреженности в матрице весов $\boldsymbol{U}$, мы можем реализовать такие структуры машин Больцмана, как ОМБ или ГМБ, с разным числом слоев. Для этого нужно разбить $\boldsymbol{x}$ на видимые и скрытые блоки и обнулить элементы $\boldsymbol{U}$, соответствующие блокам, которые не взаимодействуют. В центрированной машине Больцмана вводится вектор $\boldsymbol{\mu}$, вычитаемый из всех состояний:

$$E'(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{b}) = -(\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{U}(\boldsymbol{x} - \boldsymbol{\mu}) - (\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{b}. \tag{20.37}$$

Обычно $\boldsymbol{\mu}$ является гиперпараметром и фиксируется в начале обучения. Как правило, он выбирается, так чтобы $\boldsymbol{x} - \boldsymbol{\mu} \approx \boldsymbol{0}$ на этапе инициализации модели. Такая перепараметризация не влияет на множество распределений вероятности, представимых моделью, но изменяет динамику стохастического градиентного спуска в применении к правдоподобию. Точнее говоря, во многих случаях перепараметризация дает лучше обусловленную матрицу Гессе. В работе Melchior et al. (2013) экспериментально подтверждено, что обусловленность гессиана улучшается, и отмечено, что центрирование эквивалентно другой технике обучения машин Больцмана – **расширенному градиенту** (enhanced gradient) (Cho et al., 2011). Благодаря улучшенной обусловленности гессиана обучение может успешно завершиться даже в трудных случаях типа обучения глубокой машины Больцмана с несколькими слоями.

Другой подход к совместному обучению глубоких машин Больцмана – многопредсказательная глубокая машина Больцмана (МП-ГМБ), идея которой – рассматривать уравнения среднего поля как определение семейства рекуррентных сетей для приближенного решения любой возможной проблемы вывода (Goodfellow et al., 2013b). Вместо того чтобы обучать модель максимизации правдоподобия, мы обучаем ее, так чтобы каждая рекуррентная сеть получала верный ответ на соответствующую проблему вывода. Процесс обучения показан на рис. 20.5. Он состоит из трех частей: случайная выборка обучающего примера, случайная выборка подмножества входов сети вывода и обучение сети вывода предсказывать значения остальных блоков.

Этот общий принцип обратного распространения по графу вычислений для приближенного вывода применялся и к другим моделям (Stoyanov et al., 2011; Brakel et al., 2013). И в этих моделях, и в МП-ГМБ окончательная потеря не является нижней границей правдоподобия, а обычно основана на приближенном условном распределении отсутствующих значений, индуцируемом сетью приближенного вывода. Это значит, что в обоснование обучения таких моделей выдвигаются чисто эвристические аргументы. Если исследовать распределение $p(\boldsymbol{v})$, представленное машиной Больцмана, которая была обучена как МП-ГМБ, то оно окажется несовершенным в том смысле, что выборка по Гиббсу дает плохие примеры.

У обратного распространения по графу вывода есть два основных преимущества. Во-первых, он обучает модель так, как она реально используется, – с помощью при-

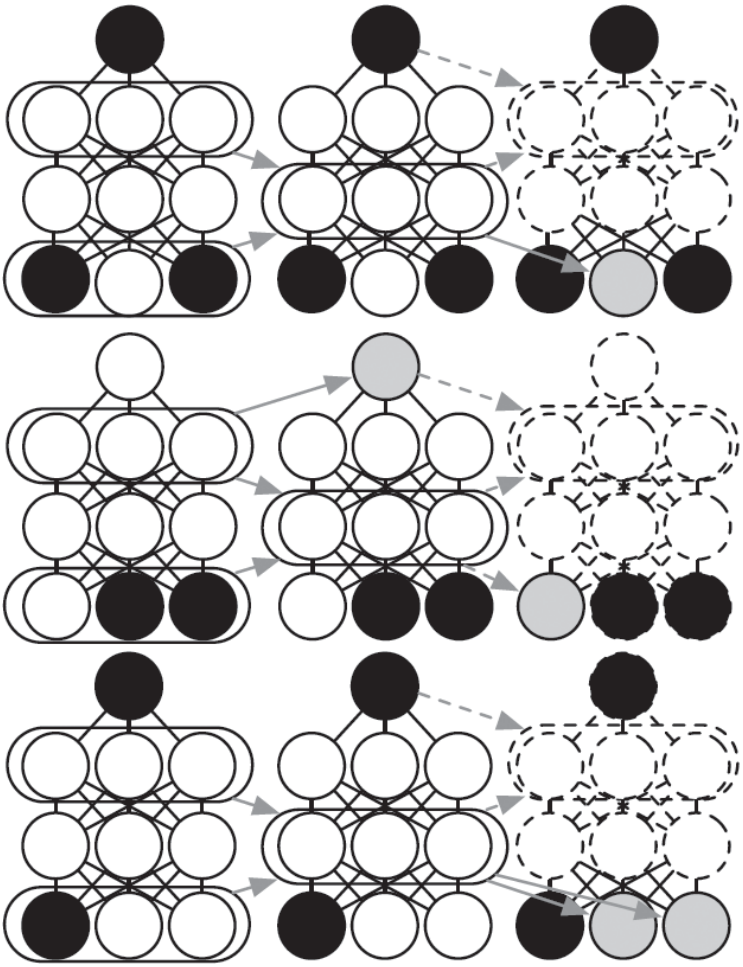

Рис. 20.5 ❖ Иллюстрация многопредсказательного процесса обучения глубокой машины Больцмана. В строках показаны разные примеры из мини-пакета для одного и того же шага обучения, а в столбцах – временной шаг процесса вывода среднего поля. Для каждого примера мы выбираем подмножество переменных, которое будет служить входом для процесса вывода. Эти переменные закрашены черным, чтобы показать обусловливание. Затем выполняется процесс вывода среднего поля, стрелки показывают влияние одних переменных на другие. В практических приложениях среднее поле разворачивается на несколько шагов, здесь же таких шагов всего два. Штриховые стрелки означают, что процесс можно было бы развернуть еще на несколько шагов. Переменные, которые не подавались на вход процесса вывода, становятся метками, они закрашены серым цветом. Процесс вывода для каждого примера можно рассматривать как рекуррентную сеть. Мы пользуемся градиентным спуском и обратным распространением, чтобы обучить эти рекуррентные сети порождать правильные метки при известных входах. Тем самым процесс среднего поля для МП-ГМБ обучается давать верные оценки. Рисунок взят из работы Goodfellow et al. (2013b) и немного модифицирован

ближенного вывода. Это означает, что приближенный вывод с целью, например, восполнить отсутствующие входные данные или выполнить классификацию, несмотря на отсутствие части данных, будет более верным при использовании МП-ГМБ, чем оригинальной ГМБ. Оригинальная ГМБ не является верным классификатором сама по себе; наилучшие результаты достигаются, когда на признаках, извлеченных ГМБ, обучается отдельный классификатор, а не когда вывод применяется для вычисления распределения меток классов. Вывод среднего поля в МП-ГМБ хорошо работает в роли классификатора даже без специальных модификаций. Второе преимущество обратного распространения по графу приближенного вывода состоит в том, что вычисляется точный градиент потери. Для оптимизации это лучше, чем приближенные градиенты, вычисляемые алгоритмом стохастической максимизации правдоподобия, которые подвержены как смещению, так и дисперсии. По всей видимости, это объясняет, почему МП-ГМБ можно обучать совместно, тогда как для ГМБ требуется жадное послойное предобучение. Недостаток обратного распространения по графу приближенного вывода – в том, что оно не позволяет оптимизировать логарифмическое правдоподобие, а дает лишь эвристическую аппроксимацию обобщенного псевдоправдоподобия.

МП-ГМБ вдохновила на создание NADE-k (Raiko et al., 2014) – расширения каркаса NADE, описанного в разделе 20.10.10.

Существуют связи между МП-ГМБ и прореживанием. Прореживание означает разделение параметров между несколькими графами вычислений, различие между которыми заключается в том, включает граф некоторый блок или нет. В МП-ГМБ параметры также разделяются между графами вычислений. Но различие между графами состоит в том, наблюдается некоторый входной блок или нет. Если блок не наблюдается, то МП-ГМБ, в отличие от прореживания, не удаляет его полностью, а рассматривает как латентную переменную, подлежащую выводу. Можно было бы представить себе применение прореживания к МП-ГМБ посредством удаления некоторых блоков, вместо того чтобы делать их латентными.

20.5. Машины Больцмана для вещественных данных

Первоначально машины Больцмана разрабатывались для бинарных данных, но во многих приложениях, в т. ч. при моделировании изображений и звука, необходимо представлять распределения вероятности вещественных значений. В некоторых случаях вещественные данные на отрезке [0, 1] можно рассматривать как представление математического ожидания бинарной случайной величины. Например, в работе Hinton (2000) полутоновые изображения в обучающем наборе рассматриваются как определение вероятностей из диапазона [0,1]. Каждый пиксель определяет вероятность того, что бинарная величина принимает значение 1, и все бинарные пиксели выбираются независимо друг от друга. Это распространенная процедура вычисления бинарных моделей для наборов полутоновых изображений. Тем не менее с теоретической точки зрения этот подход не слишком хорош, а независимо выбранные таким способом бинарные изображения напоминают шум. В этом разделе мы опишем машины Больцмана, определяющие плотность вероятности вещественных данных.

20.5.1. ОМБ Гаусса–Бернулли

Ограниченные машины Больцмана можно разработать для многих экспоненциальных семейств условных распределений (Welling et al., 2005). Наиболее распространены ОМБ с бинарными скрытыми и вещественными видимыми блоками и нормальным условным распределением видимых блоков, среднее значение которого является функцией скрытых блоков.

Существует много способов параметризации ОМБ Гаусса–Бернулли. В частности, можно выбрать, использовать для нормального распределения ковариационную матрицу или матрицу точности. Ниже будет описана формулировка с матрицей точности. Переформулирование с ковариационной матрицей не составляет труда. Мы хотим иметь условное распределение

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1}). \tag{20.38}$$

Мы можем найти, какие члены следует прибавить к функции энергии, раскрыв ненормированное логарифмическое условное распределение:

$$\log \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta}^{-1}) = -{}^1\!/_2(\boldsymbol{v} - \boldsymbol{W}\boldsymbol{h})^\top \boldsymbol{\beta}(\boldsymbol{v} - \boldsymbol{W}\boldsymbol{h}) + f(\boldsymbol{\beta}). \tag{20.39}$$

Здесь f инкапсулирует все члены, являющиеся функцией только параметров, а не случайных величин модели. Мы можем отбросить f, поскольку ее единственная роль – нормировать распределение, но эту роль сыграет статистическая сумма выбранной нами функции энергии.

Если мы включим все содержащие $\boldsymbol{v}$ члены (с противоположным знаком) уравнения (20.39) в нашу функцию энергии и не будем прибавлять никаких других членов, содержащих $\boldsymbol{v}$, то функция энергии будет представлять желаемое условное распределение $p(\boldsymbol{v} \mid \boldsymbol{h})$.

В выборе другого условного распределения $p(\boldsymbol{h} \mid \boldsymbol{v})$ нам предоставлена некоторая свобода. Заметим, что в уравнении (20.39) имеется член

$${}^1\!/_2\boldsymbol{h}^\top \boldsymbol{W}^\top \boldsymbol{\beta} \boldsymbol{W}\boldsymbol{h}. \tag{20.40}$$

Этот член нельзя включить целиком, поскольку он содержит члены вида $h_i h_j$, соответствующие ребрам между скрытыми блоками. Если бы мы включили эти члены, то получили бы линейную факторную модель, а не ограниченную машину Больцмана. При проектировании нашей машины Больцмана мы просто опускаем эти попарные произведения. При этом условное распределение $p(\boldsymbol{v} \mid \boldsymbol{h})$ не изменяется, так что уравнение (20.39) по-прежнему справедливо. Однако мы можем решить, следует ли включать члены, содержащие единственный элемент h_i. Если взять диагональную матрицу точности, то окажется, что для каждого скрытого блока h_i имеется член

$$\frac{1}{2} h_i \sum_j \beta_j W_{j,i}^2. \tag{20.41}$$

Здесь мы воспользовались тем фактом, что $h_i^2 = h_i$, потому что $h_i \in \{0, 1\}$. Если включить этот член (с противоположным знаком) в функцию энергии, то у h_i появится естественная тенденция (выражаемая смещением) к выключению, когда велики веса связей этого блока с видимыми блоками высокой точности. Решение о том, включать этот член смещения или нет, не влияет на семейство распределений, представимых моделью (в предположении, что включены параметры смещения для скрытых бло-

ков), но влияет на динамику обучения модели. Благодаря его включению активации скрытых блоков, возможно, останутся разумными даже тогда, когда абсолютные величины весов быстро возрастают.

Вот одно из возможных определений функции энергии для ОМБ Гаусса–Бернулли:

$$E(\boldsymbol{v}, \boldsymbol{h}) = {}^{1}/_{2}\boldsymbol{v}^{\top}(\boldsymbol{\beta} \odot \boldsymbol{v}) - (\boldsymbol{v} \odot \boldsymbol{\beta})^{\top}\boldsymbol{W}\boldsymbol{h} - \boldsymbol{b}^{\top}\boldsymbol{h}, \tag{20.42}$$

но можно также добавить дополнительные члены или параметризовать энергию в терминах дисперсии, а не точности.

В этом выводе не включен член смещения для видимых блоков, но его легко добавить. И последний способ модифицировать параметризацию ОМБ Гаусса–Бернулли – решить, как трактовать матрицу точности. Она может быть фиксированной (скажем, взять оценку на основе маргинальной точности данных) или обучаемой. Она может быть равна произведению тождественной матрицы на скаляр или произвольной диагональной матрицей. Обычно мы не используем в этом контексте недиагональных матриц точности, потому что некоторые операции над нормальным распределением требуют обращения матрицы, а диагональная матрица обращается тривиально. В последующих разделах мы увидим, что другие виды машин Больцмана допускают моделирование ковариационной структуры с применением различных приемов, позволяющих избежать обращения матрицы точности.

20.5.2. Неориентированные модели условной ковариации

Хотя гауссова ОМБ всегда была канонической энергетической моделью для вещественных данных, в работе Ranzato et al. (2010a) отмечено, что индуктивное смещение гауссовой ОМБ плохо соответствует статистическим вариациям, присутствующим в некоторых типах вещественных данных, особенно в естественных изображениях. Проблема в том, что значительная часть информационного содержимого естественных изображений заключена в ковариации между пикселями, а не в самих значениях пикселей. Другими словами, именно связи между пикселями, а не их абсолютные значения определяют полезную информацию, присутствующую в изображении. Поскольку гауссова ОМБ моделирует только условное среднее входных данных при условии скрытых блоков, она не способна уловить информацию об условной ковариации. В ответ на эту критику были предложены альтернативные модели, пытающиеся лучше учесть ковариацию вещественных данных. К их числу относится ОМБ со средним и ковариацией (mean and covariance RBM – mcRBM), модель среднего произведения *t*-распределения Стьюдента (mPoT) и ОМБ типа Spike and Slab RBM (ssRBM).

ОМБ со средним и ковариацией. В модели mcRBM скрытые блоки используются для независимого кодирования условного среднего и ковариации всех наблюдаемых блоков. Скрытый слой mcRBM разбит на две группы блоков: блоки среднего и блоки ковариации. Группа, моделирующая условное среднее, – это просто гауссова ОМБ. Вторая половина – ковариационная ОМБ (Ranzato et al., 2010a), которую часто называют cRBM; ее компоненты моделируют структуру условной ковариации, как описано ниже.

Точнее говоря, если бинарные блоки среднего обозначить $\boldsymbol{h}^{(m)}$, а бинарные блоки ковариации $\boldsymbol{h}^{(c)}$, то модель mcRBM определяется как комбинация двух функций энергии:

$$E_{\text{mc}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}) = E_{\text{m}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}) + E_{\text{c}}(\boldsymbol{x}, \boldsymbol{h}^{(c)}), \tag{20.43}$$

где E_{m} – стандартная функция энергии ОМБ Гаусса–Бернулли[1].

$$E_{\text{m}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}) = \frac{1}{2}\boldsymbol{x}^{\top}\boldsymbol{x} - \sum_j \boldsymbol{x}^{\top}\boldsymbol{W}_{:,j}h_j^{(m)} - \sum_j b_j^{(m)}h_j^{(m)}, \tag{20.44}$$

а E_{c} – функция энергии cRBM, моделирующая информацию об условной ковариации:

$$E_{\text{c}}(\boldsymbol{x}, \boldsymbol{h}^{(c)}) = \frac{1}{2}\sum_j h_j^{(c)}(\boldsymbol{x}^{\top}\boldsymbol{r}^{(j)})^2 - \sum_j b_j^{(c)}h_j^{(c)}. \tag{20.45}$$

Параметр $\boldsymbol{r}^{(j)}$ соответствует вектору весов ковариации, ассоциированному с $h_j^{(c)}$, а $\boldsymbol{b}^{(c)}$ – вектор смещений ковариации. Объединенная функция энергии определяет совместное распределение

$$p_{\text{mc}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}) = (1/Z)\exp\{-E_{\text{mc}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})\} \tag{20.46}$$

и соответствующее условное распределение наблюдений при условии $\boldsymbol{h}^{(m)}$ и $\boldsymbol{h}^{(c)}$ в виде многомерного нормального распределения:

$$p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, h^{(c)}) = \mathcal{N}\left(\boldsymbol{x}; \boldsymbol{C}_{\boldsymbol{x}|\boldsymbol{h}}^{\text{mc}}\left(\sum_j \boldsymbol{W}_{:,j}h_j^{(m)}\right), \boldsymbol{C}_{\boldsymbol{x}|\boldsymbol{h}}^{\text{mc}}\right). \tag{20.47}$$

Отметим, что ковариационная матрица $\boldsymbol{C}_{\boldsymbol{x}|\boldsymbol{h}}^{\text{mc}} = \left(\sum_j h_j^{(c)}\boldsymbol{r}^{(j)}\boldsymbol{r}^{(j)\top} + \boldsymbol{I}\right)^{-1}$ не является диагональной и что $\boldsymbol{W}$ – матрица весов, ассоциированная с моделированием условных средних с помощью гауссовой ОМБ. Обучить mcRBM методами сопоставительного расхождения или устойчивого сопоставительного расхождения трудно из-за недиагональной условной ковариационной матрицы. В методах CD и PCD требуется производить выборку из совместного распределения $\boldsymbol{x}$, $\boldsymbol{h}^{(m)}$, $\boldsymbol{h}^{(c)}$, что в стандартной ОМБ достигается путем выборки по Гиббсу из условных распределений. Однако в mcRBM для выборки из $p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$ необходимо вычислять $(\boldsymbol{C}^{\text{mc}})^{-1}$ на каждой итерации обучения. Для больших объемов наблюдений это может оказаться неподъемной вычислительной задачей. В работе Ranzato and Hinton (2010) прямой выборки из $p_{\text{mc}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$ удается избежать с помощью прямой выборки из маргинального распределения $p(\boldsymbol{x})$ гамильтоновым (гибридным) методом Монте-Карло (Neal, 1993) применительно к свободной энергии mcRBM.

Среднее произведение *t*-распределения Стьюдента. Модель среднего произведения *t*-распределения Стьюдента (mPoT) (Ranzato et al., 2010b) обобщает модель PoT (Welling et al., 2003a) примерно так же, как mcRBM обобщает cRBM. Достигается это путем включения ненулевых гауссовых средних за счет добавления скрытых блоков, как в гауссовой ОМБ. Подобно mcRBM, условное распределение наблюдений PoT является многомерным нормальным распределением (с недиагональной ковариационной матрицей), но, в отличие от mcRBM, дополнительное условное распределение скрытых блоков описывается условными независимыми гамма-распределениями. Гамма-распределение $\mathcal{G}(k, \theta)$ – это распределение вероятности положительных веще-

[1] Этот вариант функции энергии ОМБ Гаусса–Бернулли предполагает, что в данных изображения среднее всех пикселей равно нулю. В модель можно легко добавить пиксельные смещения, чтобы учесть ненулевые средние.

ственных чисел со средним $k\theta$. Для понимания основных идей модели mPoT знакомство с деталями гамма-распределения необязательно.

Функция энергии в модели mPoT имеет вид

$$E_{\text{mPoT}}(\boldsymbol{x}, \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)}) \tag{20.48}$$

$$= E_m(\boldsymbol{x}, \boldsymbol{h}^{(m)}) + \sum_j \left(h_j^{(c)} \left(1 + \frac{1}{2} (\boldsymbol{r}^{(j)\top} \boldsymbol{x})^2 \right) + (1 - \gamma_j) \log h_j^{(c)} \right), \tag{20.49}$$

где $\boldsymbol{r}^{(j)}$ – вектор весов ковариации, ассоциированный с блоком $h_j^{(c)}$, а функция $E_m(\boldsymbol{x}, \boldsymbol{h}^{(m)})$ определена, как в (20.44).

Как и в случае mcRBM, функция энергии в модели mPoT определяет многомерное нормальное распределение – такое, что условное распределение $\boldsymbol{x}$ имеет недиагональную ковариационную матрицу. Обучение модели mPoT – как и mcRBM – осложняется невозможностью выборки из нормального условного распределения с недиагональной ковариационной матрицей $p_{\text{mPoT}}(\boldsymbol{x} \mid \boldsymbol{h}^{(m)}, \boldsymbol{h}^{(c)})$, поэтому в работе Ranzato et al. (2010b) также предлагается прямая выборка из $p(\boldsymbol{x})$ гамильтоновым (гибридным) методом Монте-Карло.

Ограниченные машины Больцмана типа Spike and Slab. Ограниченные машины Больцмана типа Spike and Slab, или ssRBM (Courville et al., 2011), – еще один способ моделирования ковариационной структуры вещественных данных. По сравнению с mcRBM, они обладают тем преимуществом, что не нуждаются ни в обращении матрицы, ни в гамильтоновых методах Монте-Карло. Подобно mcRBM и mPoT, в бинарных скрытых блоках ssRBM закодирована условная ковариация между пикселями посредством использования вспомогательных вещественных переменных.

В ОМБ типа Spike and Slab есть два набора скрытых блоков: бинарные **spike**-блоки $\boldsymbol{h}$ и вещественные **slab**-блоки $\boldsymbol{s}$. Среднее значение видимых блоков при условии скрытых блоков равно $(\boldsymbol{h} \odot \boldsymbol{s})\boldsymbol{W}^\top$. Иначе говоря, каждый столбец $\boldsymbol{W}_{:,i}$ определяет компоненту, которая может встречаться во входных данных, когда $h_i = 1$. Соответствующая spike-переменная h_i определяет, присутствует ли эта компонента вообще. А соответствующая slab-переменная s_i определяет интенсивность этой компоненты, если она присутствует. Когда spike-переменная активна, соответствующая slab-переменная добавляет дисперсию к входным данным вдоль оси, определенной столбцом $\boldsymbol{W}_{:,i}$. Это позволяет моделировать ковариацию между входами. По счастью, методы CD и PCD с выборкой по Гиббсу по-прежнему применимы. Обращать матрицы не нужно.

Формально модель ssRBM определяется функцией энергии:

$$E_{\text{ss}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{h}) = -\sum_i \boldsymbol{x}^\top \boldsymbol{W}_{:,i} s_i h_i + \frac{1}{2} \boldsymbol{x}^\top \left(\boldsymbol{\Lambda} + \sum_i \boldsymbol{\Phi}_i h_i \right) \boldsymbol{x} \tag{20.50}$$

$$+ \frac{1}{2} \sum_i \alpha_i s_i^2 - \sum_i \alpha_i \mu_i s_i h_i - \sum_i b_i h_i + \sum_i \alpha_i \mu_i^2 h_i, \tag{20.51}$$

где b_i – смещение spike-блока h_i, $\boldsymbol{\Lambda}$ – диагональная матрица точности для наблюдений $\boldsymbol{x}$, $\alpha_i > 0$ – скалярный параметр точности вещественной slab-переменной s_i, а $\boldsymbol{\Phi}_i$ – неотрицательная диагональная матрица, определяющая $\boldsymbol{h}$-модулированный квадратичный штраф на $\boldsymbol{x}$. Параметр μ_i задает среднее slab-переменной s_i.

В случае, когда совместное распределение определено функцией энергии, вывести условные распределения в модели ssRBM сравнительно просто. Например, исключая

slab-переменные $\boldsymbol{s}$, получаем, что условное распределение наблюдений при условии бинарных spike-переменных $\boldsymbol{h}$ имеет вид

$$p_{\text{ss}}(\boldsymbol{x}\,|\,\boldsymbol{h}) = \frac{1}{P(\boldsymbol{h})}\frac{1}{Z}\int \exp\{-E(\boldsymbol{x},\boldsymbol{s},\boldsymbol{h})\}\,d\boldsymbol{s} \tag{20.52}$$

$$= \mathcal{N}\left(\boldsymbol{x}; \boldsymbol{C}^{\text{ss}}_{\boldsymbol{x}|\boldsymbol{h}} \sum_i \boldsymbol{W}_{:,i}\mu_i h_i, \boldsymbol{C}^{\text{ss}}_{\boldsymbol{x}|\boldsymbol{h}}\right), \tag{20.53}$$

где $\boldsymbol{C}^{\text{ss}}_{\boldsymbol{x}|\boldsymbol{h}} = (\boldsymbol{\Lambda} + \sum_i \boldsymbol{\Phi}_i h_i - \sum_i \alpha_i^{-1} h_i \boldsymbol{W}_{:,i}\boldsymbol{W}_{:,i}^{\top})^{-1}$. Последнее равенство имеет место, только если ковариационная матрица $\boldsymbol{C}^{\text{ss}}_{\boldsymbol{x}|\boldsymbol{h}}$ положительно определенная.

Фильтрация по spike-переменным означает, что истинное маргинальное распределение $\boldsymbol{h} \odot \boldsymbol{s}$ разреженное. Это не то же самое, что разреженное кодирование, где выборки из модели «почти никогда» (в смысле теории меры) не содержат нулей в коде и требуется, чтобы MAP-вывод индуцировал разреженность.

Если сравнить ssRBM с mcRBM и mPoT, то окажется, что ssRBM параметризует условную ковариацию между наблюдениями совершенно иначе. И mcRBM, и mPoT моделируют структуру в виде $\left(\sum_j h_j^{(c)} \boldsymbol{r}^{(j)}\boldsymbol{r}^{(j)\top} + \boldsymbol{I}\right)^{-1}$, используя активацию скрытых блоков $\boldsymbol{h}_j > 0$, чтобы наложить ограничения на условную ковариацию в направлении $\boldsymbol{r}^{(j)}$. Что же касается ssRBM, то она задает условную ковариацию между наблюдениями с помощью скрытых spike-активаций $h_i = 1$, чтобы стянуть матрицу активации вдоль направления, определяемого соответствующим весовым вектором. Условная ковариация в модели ssRBM похожа на даваемую другой моделью: анализом произведения вероятностных главных компонент (product of probabilistic principal components analysis – PoPPCA) (Williams and Agakov, 2002). В сверхполной конфигурации разреженная активация с ssRBM-параметризацией допускает значительную дисперсию (выше номинальной, определяемой матрицей $\boldsymbol{\Lambda}^{-1}$) только в избранных направлениях разреженно активированных h^i. В моделях mcRBM и mPoT сверхполное представление означало бы, что для улавливания вариативности в конкретном направлении в пространстве наблюдений потенциально пришлось бы удалить все ограничения с положительной проекцией на это направление. Отсюда следует, что эти модели хуже приспособлены к сверхполной конфигурации.

Основной недостаток ограниченной машины Больцмана типа Spike and Slab – в том, что при некоторых конфигурациях параметров получающаяся ковариационная матрица не является положительно определенной. В этом случае значения, далекие от среднего, получают большую ненормированную вероятность, так что интеграл по всем возможным исходам расходится. Обычно этой проблемы можно избежать с помощью простых эвристических приемов. Теоретически строгого решения пока не найдено. Применить ограниченную оптимизацию, чтобы явно избежать областей, где вероятность не определена, трудно, не впадая в грех чрезмерной консервативности, из-за чего может случиться так, что модель никогда не попадет в области пространства параметров, где достигается хорошее качество.

Качественно сверточные варианты ssRBM дают прекрасные примеры естественных изображений. Некоторые из них показаны на рис. 16.1.

У ssRBM есть несколько обобщений. Если включить взаимодействия высшего порядка и пулинг с усреднением по slab-переменным (Courville et al., 2014), то модель сможет обучиться отличным признакам для классификатора в случае, когда помечен-

ных данных мало. Добавление в функцию энергии члена, предотвращающего неопределенность статистической суммы, приводит к модели разреженного кодирования типа Spike and Slab (Goodfellow et al., 2013d), или S3C (spike and slab sparse coding).

20.6. Сверточные машины Больцмана

Как отмечалось в главе 9, входные данные очень высокой размерности, например изображения, предъявляют жесткие требования к моделям машинного обучения с точки зрения объема вычислений, потребной памяти и статистических свойств. Замена умножения матриц дискретной сверткой с небольшим ядром – стандартный способ решения этих проблем для входных данных с пространственной или временной структурой, инвариантной относительно параллельных переносов. В работе Desjardins and Bengio (2008) показано, что этот подход хорошо работает в применении к ОМБ.

В глубоких сверточных сетях обычно требуется операция пулинга, так что пространственный размер каждого последующего слоя меньше размера предыдущего. В сверточных сетях прямого распространения часто в качестве функции пулинга берут, например, максимум агрегируемых элементов. Не ясно, как обобщить эту идею на энергетические модели. Можно было бы ввести бинарный блок пулинга p по n бинарным детекторным блокам $\mathbf{d}$ и гарантировать, что $p = \max_i d_i$, полагая функцию энергии равной ∞ всюду, где это ограничение нарушается. Это решение плохо масштабируется, поскольку требует рассмотрения 2^n конфигураций энергии, чтобы вычислить нормировочную постоянную. Для небольшой области пулинга размера 3×3 придется выполнить $2^9 = 512$ вычислений функции энергии на один блок пулинга!

В работе Lee et al. (2009) для решения этой проблемы разработан **метод вероятностного max-пулинга** (не путайте со «стохастическим пулингом» – методом неявного построения ансамблей сверточных сетей прямого распространения). Стратегия заключается в том, чтобы наложить ограничение на детекторные блоки: не более одного активного в каждый момент времени. Это означает, что всего имеется лишь $n + 1$ состояний (по одному для случаев, когда включен один из n детекторных блоков, плюс дополнительное состояние, в котором все детекторные блоки выключены). Блок пулинга включен тогда и только тогда, когда включен один из детекторных блоков. Состоянию, в котором все блоки выключены, назначается нулевая энергия. Можно считать это описанием модели с одной переменной, имеющей $n + 1$ состояний, или, эквивалентно, модели с $n + 1$ переменными, которая назначает энергию ∞ всем совместным комбинациям переменных, кроме $n + 1$.

При всей своей эффективности вероятностный max-пулинг делает детекторные блоки взаимно исключающими, что в одних контекстах может считаться полезным регуляризирующим ограничением, а в других вредным ограничением на емкость модели. Этот метод не поддерживает пересекающихся областей пулинга, которые обычно нужны для достижения оптимального качества сверточных сетей прямого распространения, так что это ограничение, вероятно, сильно снижает качество сверточных машин Больцмана.

В работе Lee et al. (2009) продемонстрировано, что вероятностный max-пулинг можно было бы использовать для построения сверточных машин Больцмана[1]. Эта

[1] Описанная в этой работе модель названа «глубокой сетью доверия», но поскольку ее можно охарактеризовать как строго неориентированную модель с вычислимыми послойными обновлениями неподвижной точки среднего поля, то лучше было бы назвать ее глубокой машиной Больцмана.

модель умеет выполнять такие операции, как восполнение отсутствующих частей данных. Несмотря на интеллектуальную привлекательность, работать с этой моделью на практике трудно, и обычно в роли классификатора она показывает худшие результаты, чем традиционные сверточные сети, обученные с учителем.

Многие сверточные модели одинаково хорошо работают с входными данными разного пространственного размера. Для машин Больцмана изменить размер входа сложно по нескольким причинам. При изменении размера входа меняется статистическая сумма. Кроме того, во многих сверточных сетях инвариантность относительно размера достигается путем увеличения размера областей пулинга пропорционально размеру входа, но масштабировать области пулинга в машине Больцмана неудобно. В традиционных сверточных нейронных сетях можно использовать фиксированное число блоков пулинга и динамически увеличивать их размер. В машинах Больцмана большие области пулинга обходятся слишком дорого при наивном подходе. Примененный в работе Lee et al. (2009) подход – сделать детекторные блоки в одной области пулинга взаимно исключающими – решает вычислительные проблемы, но все равно не позволяет иметь области пулинга переменного размера. Предположим, к примеру, что мы обучаем модель детекторных блоков, обучающихся обнаружению границ с вероятностным max-пулингом по области 2×2. Это налагает ограничение: в каждой области 2×2 может встречаться только одна граница. Если мы затем увеличим размер входного изображения на 50% в каждом направлении, то естественно ожидать, что число границ соответственно возрастет. Если же мы вместо этого увеличим на 50% размер областей пулинга в каждом направлении до 3×3, то ограничение взаимного исключения теперь говорит, что в каждой области размера 3×3 может присутствовать не более одной границы. По мере увеличения входного изображения модель генерирует границы с меньшей плотностью. Разумеется, такие проблемы возникают, только когда модель вынуждена использовать переменный размер области пулинга, чтобы выходной вектор имел фиксированный размер. Модели с вероятностным max-пулингом все же могут принимать изображения переменного размера, при условии что карта признаков на выходе модели может масштабироваться пропорционально размеру входного изображения.

Пиксели на границе изображения тоже представляют сложность, усугубляющуюся тем фактом, что связи в машине Больцмана симметричны. Если мы не будем неявно дополнять вход нулями, то скрытых блоков будет меньше, чем видимых, и видимые блоки на границе изображения будут моделироваться плохо, потому что принадлежат рецептивному полю меньшего числа скрытых блоков. Но если производить неявное дополнение нулями, то скрытые блоки на границе будут управляться меньшим числом входных пикселей, так что активация может не произойти, когда необходимо.

20.7. Машины Больцмана для структурных и последовательных выходов

В случае структурного выхода мы хотим обучить модель, умеющую отображать вход $\boldsymbol{x}$ на выход $\boldsymbol{y}$, так что различные элементы $\boldsymbol{y}$ связаны друг с другом и должны подчиняться некоторым ограничениям. Например, в задаче синтеза речи $\boldsymbol{y}$ – звуковой сигнал, и полный выходной сигнал должен звучать как связная речь.

Естественный способ представить связи между элементами $\boldsymbol{y}$ – воспользоваться распределением вероятности $p(\mathbf{y} \mid \boldsymbol{x})$. Такую вероятностную модель могут предложить машины Больцмана, обобщенные на моделирование условных распределений.

Тот же инструментарий условного моделирования с помощью машины Больцмана можно применить не только к задаче структурного вывода, но и для моделирования последовательностей. В этом случае вместо отображения входа $\boldsymbol{x}$ на выход $\boldsymbol{y}$ модель должна оценить распределение вероятности последовательности переменных $p(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\tau)})$. Для решения этой задачи условные машины Больцмана могут представить факторы виды $p(\mathbf{x}^{(t)} \mid \mathbf{x}^{(1)}, ..., \mathbf{x}^{(t-1)})$.

Важная для киноиндустрии и видеоигр задача – смоделировать последовательности углов сочленения суставов в скелетах, используемых для отрисовки трехмерных персонажей. Эти последовательности часто запоминаются системами захвата движения при регистрации движений актеров. Вероятностная модель движения персонажа позволяет генерировать новые, не встречавшиеся ранее, но реалистичные движения. Для решения этой задачи в работе Taylor et al. (2007) предложено моделирование условной ОМБ $p(\boldsymbol{x}^{(t)} \mid \boldsymbol{x}^{(t-1)}, ..., \boldsymbol{x}^{(t-m)})$ для малых m. Модель представляет собой ОМБ над распределением $p(\boldsymbol{x}^{(t)})$, в которой параметры смещения – линейная функция от предыдущих m значений $\boldsymbol{x}$. При обусловливании разными значениями $\boldsymbol{x}^{(t-1)}$ и более ранних переменных мы получаем новую ОМБ над $\mathbf{x}$. Веса в ОМБ над $\mathbf{x}$ никогда не меняются, но за счет обусловливания по различным прошлым значениям мы можем изменять вероятность активности различных скрытых блоков ОМБ. Активируя и деактивируя различные подмножества скрытых блоков, мы можем вносить значительные изменения в индуцированное распределение вероятности $\mathbf{x}$. Возможны и другие варианты условной ОМБ (Mnih et al., 2011) и моделирования последовательностей с помощью условных ОМБ (Taylor and Hinton, 2009; Sutskever et al., 2009; Boulanger-Lewandowski et al., 2012).

Еще одна задача – моделирование распределения последовательностей музыкальных нот для создания песен. В работе Boulanger-Lewandowski et al. (2012) предложена модель последовательности **РНС-ОМБ** (англ. RNN-RBM), которая применена к этой задаче. Это порождающая модель последовательности кадров $\boldsymbol{x}^{(t)}$, состоящая из РНС, которая порождает параметры ОМБ для каждого временного шага. В отличие от предыдущих подходов, где от шага к шагу варьировались только параметры смещения ОМБ, РНС, используемая в этой модели, порождает все параметры ОМБ, включая и веса. Для обучения модели мы должны выполнить обратное распространение градиента функции потерь по РНС. Функция потерь применяется не напрямую к выходам РНС, а к ОМБ. Это означает, что мы должны приближенно продифференцировать потерю по параметрам ОМБ, применив метод сопоставительного расхождения или другой похожий алгоритм. Затем приближенный градиент можно обратно распространить по РНС, применив обычный алгоритм обратного распространения во времени.

20.8. Другие машины Больцмана

Существует еще много вариантов машин Больцмана.

Для обобщения машин Больцмана можно применять различные критерии обучения. Мы сосредоточили внимание на машинах Больцмана, обучаемых приближенно максимизировать порождающий критерий $\log p(\boldsymbol{v})$. Можно вместо этого обучить дискриминантную ОМБ, нацеленную на максимизацию $\log p(y \mid \boldsymbol{v})$ (Larochelle and Bengio, 2008). Этот подход часто дает наилучшие результаты, если используется линейная комбинация порождающего и дискриминантного критериев. К сожалению,

ОМБ не так хорошо обучаются с учителем, как МСП, по крайней мере с применением существующих технологий.

В большинстве практически используемых машин Больцмана функция энергии включает только взаимодействия второго порядка, т. е. представляет собой суммы большого числа членов, каждый из которых является произведением всего двух случайных величин, например $v_i W_{i,j} h_j$. Можно обучить и машины Больцмана более высокого порядка (Sejnowski, 1987), для которых члены функции энергии являются произведениями многих величин. Трехсторонние взаимодействия между скрытым блоком и двумя разными изображениями могут моделировать пространственное преобразование между текущим и следующим кадрами видео (Memisevic and Hinton, 2007, 2010). Умножение на унитарную переменную класса может изменить связь между видимым и текущим блоками в зависимости от того, бит какого класса поднят (Nair and Hinton, 2009). Недавний пример использования взаимодействий высшего порядка дает машина Больцмана с двумя группами скрытых блоков, одна из которых взаимодействует с видимыми блоками $\boldsymbol{v}$ и меткой класса y, а другая – только с входными значениями $\boldsymbol{v}$ (Luo et al., 2011). Это можно интерпретировать как поощрение некоторых скрытых блоков обучаться моделированию входа с использованием признаков, релевантных классу; кроме того, дополнительные скрытые блоки обучаются объяснять мелкие детали, необходимые для придания реалистичности примерам $\boldsymbol{v}$, не определяя класса примера. Еще одно использование взаимодействий высшего порядка – пропускание части признаков. В работе Sohn et al. (2013) описана машина Больцмана с взаимодействиями третьего порядка и бинарными масочными переменными, ассоциированными с каждым видимым блоком. Если масочная переменная равна нулю, то она устраняет влияние соответствующего видимого блока на скрытые. Это позволяет убирать видимые блоки, не релевантные задаче классификации, из пути вывода, на котором оценивается класс.

Вообще, инфраструктура машин Больцмана – богатое поле для исследований, где возможных структур моделей гораздо больше, чем изучено до сих пор. Для разработки нового вида машин Больцмана требуется больше тщатнльности и изобретательности, чем для разработки нового слоя нейронной сети, поскольку зачастую трудно подобрать функцию энергии, допускающую обсчет всевозможных условных распределений, которые необходимы для использования модели. Несмотря на требуемые немалые усилия, эта область остается открытой для инноваций.

20.9. Обратное распространение через случайные операции

В традиционных нейронных сетях реализовано детерминированное преобразование некоторых входных переменных $\boldsymbol{x}$. Но при разработке порождающих моделей часто желательно наделить нейронную сеть способностью к стохастическим преобразованиям $\boldsymbol{x}$. Один из способов добиться этой цели – пополнить нейронную сеть дополнительными входами $\boldsymbol{z}$, выбранными из какого-нибудь простого распределения, например равномерного или нормального. Тогда на внутреннем уровне сеть будет и дальше выполнять детерминированные вычисления, но наблюдателю, не имеющему доступа к $\boldsymbol{z}$, функция $f(\boldsymbol{x}, \boldsymbol{z})$ будет казаться стохастической. При условии что f непрерывна и дифференцируема, мы можем как обычно вычислить градиенты, необходимые для обучения методом обратного распространения.

В качестве примера рассмотрим операцию, состоящую из выборки примеров y из нормального распределения со средним μ и дисперсией σ^2:

$$\mathrm{y} \sim \mathcal{N}(\mu, \sigma^2). \tag{20.54}$$

Поскольку отдельный пример y порождается не функцией, а процессом выборки, выдающим новый результат при каждом запросе, взятие производных y по параметрам распределения μ и σ^2 может показаться противоречащим интуиции. Однако мы можем переформулировать процесс выборки как преобразование случайной величины $\mathrm{z} \sim \mathcal{N}(z; 0, 1)$ для получения примера из желаемого распределения:

$$y = \mu + \sigma z. \tag{20.55}$$

Теперь мы можем выполнить обратное распространение через операцию выборки, рассматривая ее как детерминированную операцию с дополнительным входом z. Важно, что дополнительный вход – это случайная величина, распределение которой не является функцией от любой из величин, чьи производные мы хотим вычислять. Этот результат говорит, как бесконечно малое изменение μ или σ отразилось бы на выходе, если бы мы могли повторить операцию выборки с тем же значением z.

Зная, как выполнить обратное распространение через эту операцию выборки, мы можем включить ее в объемлющий граф. Можно строить элементы графа на базе выхода выборочного распределения. Например, мы можем вычислять производные некоторой функции потери $J(y)$. Можно также строить элементы графа, выходы которых являются входами или параметрами операции выборки. Например, можно было бы построить больший граф с $\mu = f(\boldsymbol{x}; \boldsymbol{\theta})$ и $\sigma = \mathrm{g}(\boldsymbol{x}; \boldsymbol{\theta})$. В этом пополненном графе мы можем воспользоваться обратным распространением через эти функции для вычисления $\nabla_{\theta} J(y)$.

Принцип, использованный в этом примере выборки из нормального распределения, применим и в более общей ситуации. Мы можем выразить любое распределение вероятности вида $p(\mathrm{y}; \boldsymbol{\theta})$ или $p(\mathrm{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ как $p(\mathrm{y} \mid \boldsymbol{\omega})$, где $\boldsymbol{\omega}$ – переменная, содержащая как параметры $\boldsymbol{\theta}$, так и (если это осмыслено) входы $\boldsymbol{x}$. Зная значение y, выбранное из распределения $p(\mathrm{y} \mid \boldsymbol{\omega})$, где $\boldsymbol{\omega}$ может, в свою очередь, быть функцией от других переменных, мы можем переписать

$$\mathbf{y} \sim p(\mathbf{y} \mid \boldsymbol{\omega}) \tag{20.56}$$

в виде

$$\boldsymbol{y} = f(\boldsymbol{z}; \boldsymbol{\omega}), \tag{20.57}$$

где $\boldsymbol{z}$ – источник случайности. Затем можно вычислить производные $\boldsymbol{y}$ по $\boldsymbol{\omega}$ с помощью традиционных средств, например алгоритма обратного распространения в применении к f в предположении, что f непрерывна и дифференцируема почти всюду. Важно, что $\boldsymbol{\omega}$ не должна быть функцией $\boldsymbol{z}$, а $\boldsymbol{z}$ не должна быть функцией $\boldsymbol{\omega}$. Эту технику часто называют **перепараметризацией**, **стохастическим обратным распространением** или **методом малых возмущений**.

Из требования о непрерывности и дифференцируемости f, конечно, вытекает, что $\boldsymbol{y}$ должна быть непрерывна. Если мы хотим выполнять обратное распространение через процесс выборки, порождающий дискретные примеры, то все же возможно оценить градиент по $\boldsymbol{\omega}$, применяя алгоритмы обучения с подкреплением, например варианты алгоритма REINFORCE (Williams, 1992), который обсуждается в разделе 20.9.1.

В приложениях нейронных сетей мы обычно выбираем $\boldsymbol{z}$ из какого-нибудь простого распределения, например равномерного или нормального, а чтобы получить более сложные распределения, разрешаем детерминированной части сети изменять форму входа.

Идея распространения градиентов или оптимизации посредством стохастических операций восходит еще к середине XX столетия (Price, 1958; Bonnet, 1964) и впервые была применена к машинному обучению в контексте обучения с подкреплением (Williams, 1992). В более близкое к нам время она применялась к вариационным аппроксимациям (Opper and Archambeau, 2009) и к стохастическим и порождающим нейронным сетям (Bengio et al., 2013b; Kingma, 2013; Kingma and Welling, 2014b,a; Rezende et al., 2014; Goodfellow et al., 2014c). Многие сети, в т. ч. шумоподавляющие автокодировщики и сети, регуляризируемые методом прореживания, также естественно проектируются для приема шума на входе, не требуя специальной перепараметризации, для того чтобы сделать шум независимым от модели.

20.9.1. Обратное распространение через дискретные стохастические операции

Если модель выдает на выходе дискретную переменную $\boldsymbol{y}$, то перепараметризация неприменима. Предположим, что модель принимает входы $\boldsymbol{x}$ и параметры $\boldsymbol{\theta}$, инкапсулированные вектором $\boldsymbol{\omega}$, и объединяет их со случайным шумом $\boldsymbol{z}$ для порождения $\boldsymbol{y}$:

$$\boldsymbol{y} = f(\boldsymbol{z}; \boldsymbol{\omega}). \tag{20.58}$$

Поскольку $\boldsymbol{y}$ дискретна, f должна быть кусочно-постоянной функцией. Производные такой функции бесполезны во всех точках. В точках разрыва производная не определена, но это еще меньшая из бед. Настоящая беда в том, что производная равна нулю на участках постоянства, т. е. почти всюду. Поэтому производные любой функции стоимости $J(\boldsymbol{y})$ ничего не говорят о том, как обновлять параметры модели $\boldsymbol{\theta}$.

Алгоритм REINFORCE (REward Increment = nonnegative Factor × Offset Reinforcement × Characteristic Eligibility) предлагает инфраструктуру для определения семейства простых, но очень эффективных решений (Williams, 1992). Основная идея заключается в том, что хотя $J(f(\boldsymbol{z}; \boldsymbol{\omega}))$ – кусочно-постоянная функция с бесполезными производными, ожидаемая стоимость $\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})} J(f(\boldsymbol{z}; \boldsymbol{\omega}))$ часто является гладкой функцией, пригодной для градиентного спуска.

Хотя это математическое ожидание обычно вычислительно неразрешимо, если размерность $\boldsymbol{y}$ велика (или $\boldsymbol{y}$ является результатом композиции большого числа дискретных стохастических решений), для него можно получить несмещенную оценку, вычислив среднее методом Монте-Карло. Стохастическую оценку градиента можно использовать совместно с алгоритмом СГС или другим методом стохастической градиентной оптимизации.

Простейший вариант алгоритма REINFORCE получается, если просто продифференцировать ожидаемую стоимость:

$$\mathbb{E}_z[J(\boldsymbol{y})] = \sum_{\boldsymbol{y}} J(\boldsymbol{y}) p(\boldsymbol{y}), \tag{20.59}$$

$$\frac{\partial \mathbb{E}[J(\boldsymbol{y})]}{\partial \boldsymbol{\omega}} = \sum_{y} J(\boldsymbol{y}) \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.60}$$

$$= \sum_{y} J(\boldsymbol{y}) p(\boldsymbol{y}) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.61}$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)} \sim p(\boldsymbol{y}), i=1}^{m} J(\boldsymbol{y}^{(i)}) \frac{\partial \log p(\boldsymbol{y}^{(i)})}{\partial \boldsymbol{\omega}}. \tag{20.62}$$

Уравнение (20.60) опирается на предположение, что J не ссылается на $\boldsymbol{\omega}$ напрямую. Ослабить это предположение и тем самым обобщить решение очень просто. В уравнении (20.61) использовано правило дифференцирования логарифма $\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} = \frac{1}{p(\boldsymbol{y})} \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}$. Уравнение (20.62) дает несмещенную оценку градиента методом Монте-Карло.

Всюду, где в этом разделе встречается $p(\boldsymbol{y})$, можно было бы с тем же успехом написать $p(\boldsymbol{y} \mid \boldsymbol{x})$, поскольку $p(\boldsymbol{y})$ параметризовано $\boldsymbol{\omega}$, а $\boldsymbol{\omega}$ содержит $\boldsymbol{\theta}$ и $\boldsymbol{x}$, если $\boldsymbol{x}$ вообще присутствует.

Эта простая оценка по алгоритму REINFORCE обладает одним недостатком – очень высокой дисперсией, поэтому для получения хорошей оценки градиента нужно выбрать много примеров $\boldsymbol{y}$. Иначе говоря, если выбрать только один пример, то алгоритм СГС будет сходиться очень медленно и потребуется уменьшать скорость обучения. Дисперсию оценки можно значительно снизить, воспользовавшись методами **снижения дисперсии** (Wilson, 1984; L'Ecuyer, 1994). Идея в том, чтобы модифицировать оценку таким образом, что математическое ожидание остается неизменным, а дисперсия уменьшается. В контексте REINFORCE предложенные методы снижения дисперсии включают вычисление **базового значения**, которое используется для смещения $J(\boldsymbol{y})$. Отметим, что любое смещение $b(\boldsymbol{\omega})$, не зависящее от $\boldsymbol{y}$, не изменяет математического ожидания оценки градиента, потому что

$$E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] = \sum_{\boldsymbol{y}} p(\boldsymbol{y}) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.63}$$

$$= \sum_{\boldsymbol{y}} \frac{\partial p(\boldsymbol{y})}{\partial \boldsymbol{\omega}} \tag{20.64}$$

$$= \frac{\partial}{\partial \boldsymbol{\omega}} \sum_{\boldsymbol{y}} p(\boldsymbol{y}) = \frac{\partial}{\partial \boldsymbol{\omega}} 1 = 0, \tag{20.65}$$

а это означает, что

$$E_{p(\boldsymbol{y})}\left[(J(\boldsymbol{y}) - b(\boldsymbol{\omega})) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] = E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y}) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] - b(\boldsymbol{\omega}) E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right] \tag{20.66}$$

$$= E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y}) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}\right]. \tag{20.67}$$

Далее, мы можем получить оптимальное значение $b(\boldsymbol{\omega})$, вычислив дисперсию $(J(\boldsymbol{y}) - b(\boldsymbol{\omega})) \frac{\partial \log p(\boldsymbol{y})}{\partial \boldsymbol{\omega}}$ относительно распределения $p(\boldsymbol{y})$ и минимизировав его относительно $b(\boldsymbol{\omega})$. В результате мы найдем, что оптимальные базовые значения $b^*(\boldsymbol{\omega})_i$ различаются для всех элементов ω_i вектора $\boldsymbol{\omega}$:

$$b^*(\boldsymbol{\omega})_i = \frac{E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]}{E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]}. \tag{20.68}$$

Таким образом, оценка градиента по ω_i принимает вид

$$(J(\boldsymbol{y}) - b(\boldsymbol{\omega})_i)\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}, \tag{20.69}$$

где $b(\boldsymbol{\omega})_i$ оценивает приведенное выше значение $b^*(\boldsymbol{\omega})_i$. Обычно оценку b получают, добавляя новые выходы в нейронную сеть и обучая их оценивать величины $E_{p(\boldsymbol{y})}\left[J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]$ и $E_{p(\boldsymbol{y})}\left[\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2\right]$ для каждого элемента $\boldsymbol{\omega}$. Эти дополнительные выходы можно обучить, взяв в качестве целевой функции среднеквадратическую ошибку и используя соответственно $J(\boldsymbol{y})\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2$ и $\frac{\partial \log p(\boldsymbol{y})}{\partial \omega_i}^2$ в качестве целей, когда $\boldsymbol{y}$ выбирается из $p(\boldsymbol{y})$ для заданного $\boldsymbol{\omega}$. Тогда оценку b можно восстановить, подставив эти оценки в уравнение (20.68). В работе Mnih and Gregor (2014) предпочтение отдано использованию одного разделяемого (между всеми элементами ω_i) выхода, обученного с меткой $J(\boldsymbol{y})$, а в качестве базового значения берется $b(\boldsymbol{\omega}) \approx E_{p(\boldsymbol{y})}[J(\boldsymbol{y})]$.

Методы снижения дисперсии были предложены в контексте обучения с подкреплением (Sutton et al., 2000; Weaver and Tao, 2001), как обобщение предшествующей работы для случая бинарного вознаграждения Dayan (1990). Примеры современного использования алгоритма REINFORCE со сниженной дисперсией в контексте глубокого обучения см. в работах Bengio et al. (2013b), Mnih and Gregor (2014), Ba et al. (2014), Mnih et al. (2014), Xu et al. (2015). Помимо использования зависящего от входа базового значения $b(\boldsymbol{\omega})$, в работе Mnih and Gregor (2014) установлено, что масштаб $(J(\boldsymbol{y}) - b(\boldsymbol{\omega}))$ можно регулировать во время обучения путем деления на его стандартное отклонение, оцененное с помощью скользящего среднего; получается своего рода адаптивная скорость обучения, которая противостоит эффекту важных вариаций абсолютной величины этого значения, имеющих место в процессе обучения. Авторы назвали эту технику эвристической **нормировкой дисперсии**.

Основанные на алгоритме REINFORCE оценки можно интерпретировать как оценивание градиента путем коррелирования выбора $\boldsymbol{y}$ с соответствующими значениями $J(\boldsymbol{y})$. Если хорошее значение $\boldsymbol{y}$ при текущей параметризации маловероятно, то может потребоваться много времени на то, чтобы случайно получить его и необходимый сигнал о том, что эту конфигурацию следует подкрепить.

20.10. Ориентированные порождающие сети

Как отмечалось в главе 16, ориентированные графические модели составляют важный класс графических моделей. Но, несмотря на их популярность в широком сообществе машинного обучения, в более узком кругу специалистов по глубокому обучению примерно до 2013 года их затмевали неориентированные модели типа ОМБ.

В этом разделе мы рассмотрим некоторые стандартные графические модели, которые традиционно ассоциируются с глубоким обучением. Мы уже описывали глубокие сети доверия, представляющие собой частично ориентированную модель. Мы также описывали модели разреженного кодирования, которые можно считать мелкими ориентированными порождающими моделями. В контексте глубокого обучения они часто используются для обучения признакам, хотя оставляют желать лучшего как метод генерации примеров и оценивания плотности. Теперь мы опишем различные виды глубоких полностью ориентированных моделей.

20.10.1. Сигмоидные сети доверия

Сигмоидная сеть доверия (Neal, 1990) – простой вид ориентированной графической модели со специфическим условным распределением вероятности. В общем случае такую сеть можно представлять себе как вектор бинарных состояний $\boldsymbol{s}$, каждый элемент которого зависит от своих предков:

$$p(s_i) = \sigma\left(\sum_{j<i} W_{j,i} s_j + b_i\right). \tag{20.70}$$

В самом распространенном случае сигмоидная сеть доверия состоит из большого числа слоев, а предковая выборка проходит через много скрытых слоев и в конечном итоге генерирует видимый слой. Эта структура очень похожа на глубокую сеть доверия – с тем отличием, что блоки в начале процесса выборки независимы друг от друга, а не выбираются из ограниченной машины Больцмана. Такая структура интересна по целому ряду причин, в частности потому, что является универсальным аппроксиматором распределений вероятности видимых блоков в том смысле, что может аппроксимировать любое распределение вероятности бинарных величин с произвольной точностью при наличии достаточно большого числа слоев, даже если ширина каждого слоя ограничена размерностью видимого слоя (Sutskever and Hinton, 2008).

Хотя в сигмоидной сети доверия генерация выборки видимых слоев очень эффективна, о большинстве других операций этого не скажешь. Вывод скрытых блоков при условии видимых блоков вычислительно неразрешим. Как и вывод среднего поля, поскольку для вычисления вариационной нижней границы нужно знать математические ожидания клик, а они охватывают слои целиком. Из-за трудности этой проблемы ориентированные дискретные сети не получили широкого распространения.

Один из подходов к выводу в сигмоидной сети доверия – построить другую нижнюю границу специально для таких сетей (Saul et al., 1996). Но этот подход применялся только к совсем небольшим сетям. Другое решение – воспользоваться механизмами обученного вывода из раздела 19.5. Машина Гельмгольца (Dayan et al., 1995; Dayan and Hinton, 1996) – это сигмоидная сеть доверия в сочетании с сетью вывода, предсказывающей параметры распределения среднего поля скрытых блоков. В современных подходах к сигмоидным сетям доверия (Gregor et al., 2014; Mnih and Gregor, 2014) по-прежнему используется эта идея сети вывода. Но все эти методы остаются трудными вследствие дискретной природы латентных переменных. Нельзя просто выполнить обратное распространение через выход сети вывода, приходится вместо этого использовать относительно ненадежные механизмы обратного распространения через дискретные процессы выборки, как описано в разделе 20.9.1. Недавние подходы на основе выборки по значимости, алгоритма бодрствования-сна с изменением

весом (Bornschein and Bengio, 2015) и двусторонних машин Гельмгольца (Bornschein et al., 2015) сделали возможным быстрое обучение сигмоидных сетей доверия и на эталонных задачах достигли качества, не уступающего лучшим образцам.

Частным случаем сигмоидных сетей доверия являются сети без латентных переменных. В этом случае обучение эффективно, поскольку нет нужды исключать латентные переменных из правдоподобия. Так называемые авторегрессивные сети обобщают эту сеть доверия с полной видимостью на другие виды переменных, помимо бинарных, и на другие структуры условных распределений, помимо лог-линейных связей. Авторегрессивные сети описаны в разделе 20.10.7.

20.10.2. Дифференцируемые генераторные сети

В основе многих порождающих моделей лежит идея использования дифференцируемой **генераторной сети**. Модель преобразует примеры латентных переменных **z** в примеры **x** или в распределения примеров **x**, применяя дифференцируемую функцию $g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$, которая обычно представляется нейронной сетью. В этот класс моделей входят автокодировщики, которые объединяют генераторную сеть с сетью вывода, порождающие состязательные сети, которые объединяют генераторную сеть с дискриминантной, и методы, в которых генераторные сети используются сами по себе.

По сути своей генераторные сети – это просто параметризованные вычислительные процедуры для генерации примеров, где архитектура предоставляет семейство распределений, из которых можно производить выборку, а с помощью параметров выбирается конкретное распределение из этого семейства.

Например, стандартная процедура выборки из нормального распределения со средним μ и ковариационной матрицей $\boldsymbol{\Sigma}$ заключается в том, чтобы подать выборку $\boldsymbol{z}$ из нормального распределения с нулевым средним и единичной ковариационной матрицей на вход очень простой генераторной сети, которая содержит всего один аффинный слой:

$$\boldsymbol{x} = g(\boldsymbol{z}) = \mu + \boldsymbol{L}\boldsymbol{z}, \tag{20.71}$$

где $\boldsymbol{L}$ определяется разложением Холеского матрицы $\boldsymbol{\Sigma}$.

Генераторы псевдослучайных чисел также могут использовать нелинейные преобразования простых распределений. Например, в **методе обратного преобразования** (Devroye, 2013) выбирается скаляр z из распределения $U(0, 1)$ и применяется нелинейное преобразование к скаляру x. В этом случае $g(z)$ определяется как обращение интегральной функции распределения $F(x) = \int_{-\infty}^{x} p(v)dv$. Если мы умеем задавать $p(x)$, интегрировать по x и обращать получающуюся функцию, то сможем произвести выборку из $p(x)$ без применения машинного обучения.

Чтобы сгенерировать примеры из более сложных распределений, которые трудно описать непосредственно, трудно проинтегрировать или трудно обратить результат интегрирования, мы пользуемся сетью прямого распространения для представления параметрического семейства нелинейных функций g и с помощью обучающих данных выводим параметры, отбирающие нужную функцию.

Можно считать, что g задает нелинейную замену переменных, преобразующую распределение **z** в желаемое распределение **x**.

Напомним (см. формулу 3.47), что для обратимой дифференцируемой непрерывной функции g имеет место тождество

$$p_z(\boldsymbol{z}) = p_x(g(\boldsymbol{z}))\left|\det\left(\frac{\partial g}{\partial \boldsymbol{z}}\right)\right|. \tag{20.72}$$

Тем самым мы неявно определяем распределение вероятности $\mathbf{x}$:

$$p_x(\boldsymbol{x}) = \frac{p_z(g^{-1}(\boldsymbol{x}))}{\left|\det\left(\frac{\partial g}{\partial \boldsymbol{z}}\right)\right|}. \tag{20.73}$$

Разумеется, при некоторых g это выражение трудно вычислить, поэтому мы часто применяем непрямые методы обучения g, вместо того чтобы пытаться максимизировать $\log p(\boldsymbol{x})$ непосредственно.

В некоторых случаях мы используем g не для получения примера $\boldsymbol{x}$ напрямую, а для определения условного распределения $\boldsymbol{x}$. Например, можно было бы воспользоваться генераторной сетью, последний слой которой состоит из сигмоидных выходов, для получения параметров распределений Бернулли:

$$p(\mathrm{x}_i = 1 \mid \boldsymbol{z}) = g(\boldsymbol{z})_i. \tag{20.74}$$

В этом случае, используя g для определения $p(\boldsymbol{x} \mid \boldsymbol{z})$, мы определяем распределение $\boldsymbol{x}$ путем исключения $\boldsymbol{z}$:

$$p(\boldsymbol{x}) = \mathbb{E}_z p(\boldsymbol{x} \mid \boldsymbol{z}). \tag{20.75}$$

Оба подхода определяют распределение $p_g(\boldsymbol{x})$ и позволяют обучать различные критерии p_g, используя технику перепараметризации, описанную в разделе 20.9.

У двух разных подходов к определению генераторных сетей – порождение параметров условного распределения и прямое порождение примеров – есть свои плюсы и минусы. Если генераторная сеть определяет условное распределение $\boldsymbol{x}$, то она способна генерировать как дискретные, так и непрерывные данные. Если же генераторная сеть порождает примеры непосредственно, то она может генерировать только непрерывные данные (можно было бы включить дискретизацию в прямое распространение, но тогда модель нельзя было бы обучить с помощью обратного распространения). Преимущество непосредственной выборки – в том, что мы больше не ограничены просто записываемыми условными распределениями, к которым проектировщик мог бы легко применять алгебраические преобразования.

В обоснование подходов, основанных на дифференцируемых генераторных сетях, выдвигается успешное применение градиентного спуска к дифференцируемым сетям прямого распространения для целей классификации. В контексте обучения с учителем глубокие сети прямого распространения, обученные градиентными методами, практически наверняка приводят к успеху при наличии достаточного числа скрытых блоков и обучающих данных. Нельзя ли перенести тот же рецепт успеха на порождающее моделирование?

Порождающее моделирование выглядит труднее классификации или регрессии, потому что процесс обучения требует оптимизации неразрешимых критериев. В дифференцируемых генераторных сетях критерии неразрешимы, потому что данные не содержат одновременно входов $\boldsymbol{z}$ и выходов $\boldsymbol{x}$ генераторной сети. В случае обучения с учителем задавались входы $\boldsymbol{x}$ и выходы $\boldsymbol{y}$, а процедуре оптимизации нужно было только обучиться, как порождать заданное отображение. В случае порождающего

моделирования процедура обучения должна определить, как организовать пространство $\boldsymbol{z}$ полезным способом и, кроме того, как отобразить $\boldsymbol{z}$ в $\boldsymbol{x}$.

В работе Dosovitskiy et al. (2015) изучалась более простая задача – когда соответствие между $\boldsymbol{z}$ и $\boldsymbol{x}$ задано. Точнее говоря, обучающие данные представляют собой сгенерированные компьютером изображения стула. Латентные переменные $\boldsymbol{z}$ – параметры движка отрисовки, определяющие выбор модели стула, его положение и другие детали, влияющие на генерацию изображения. Используя эти синтетические данные, сверточная сеть может обучиться отображать $\boldsymbol{z}$ (описания содержимого изображения) в $\boldsymbol{x}$ (аппроксимации отрисованных изображений). Это наводит на мысль, что современные дифференцируемые генераторные сети обладают достаточной емкостью, чтобы стать хорошими порождающими моделями, и что современные алгоритмы оптимизации вполне способны их аппроксимировать. Трудность заключается в том, как обучать генераторные сети, если значение $\boldsymbol{z}$, соответствующее каждому $\boldsymbol{x}$, не фиксировано и заранее неизвестно.

В следующих разделах описано несколько подходов к обучению дифференцируемых генераторных сетей при наличии только обучающих примеров $\boldsymbol{x}$.

20.10.3. Вариационные автокодировщики

Вариационный автокодировщик, или VAE (Kingma, 2013; Rezende et al., 2014), – это ориентированная модель, в которой применяется обученный приближенный вывод и которую можно обучить с помощью одних лишь градиентных методов.

Для порождения выборки из модели VAE сначала выбирает пример $\boldsymbol{z}$ из кодового распределения $p_{\text{model}}(\boldsymbol{z})$. Затем этот пример прогоняется через дифференцируемую генераторную сеть $g(\boldsymbol{z})$. Наконец, производится выборка $\boldsymbol{x}$ из распределения $p_{\text{model}}(\boldsymbol{x}, g(\boldsymbol{z})) = p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z})$. Но на этапе обучения для получения $\boldsymbol{z}$ используется сеть приближенного вывода (или кодировщик) $q(\boldsymbol{z} \mid \boldsymbol{x})$, и тогда $p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z})$ рассматривается как декодирующая сеть.

Главная идея вариационных автокодировщиков заключается в том, что их можно обучить с помощью максимизации вариационной нижней границы $\mathcal{L}(q)$, ассоциированной с точкой $\boldsymbol{x}$:

$$\mathcal{L}(q) = \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}\mid\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x}) + \mathcal{H}(q(\mathbf{z} \mid \boldsymbol{x})) \tag{20.76}$$

$$= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}\mid\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z}) - D_{\text{KL}}(q(\mathbf{z} \mid \boldsymbol{x}) \| p_{\text{model}}(\mathbf{z})) \tag{20.77}$$

$$\leq \log p_{\text{model}}(\boldsymbol{x}). \tag{20.78}$$

В равенстве (20.76) первый член – не что иное, как совместное логарифмическое правдоподобие видимых и латентных переменных относительно приближенного апостериорного распределения латентных переменных (как и в EM-алгоритме, с тем исключением, что здесь используется приближенное, а не точное апостериорное распределение). Если в качестве q выбрано нормальное распределение с шумом, добавленным к предсказанному среднему значению, то максимизация этого энтропийного члена поощряет увеличение стандартного отклонения шума. В общем случае энтропийный член поощряет вариационное апостериорное распределение отдавать больше массы вероятности многим значениям $\boldsymbol{z}$, которые могли бы породить $\boldsymbol{x}$, а не сосредоточивать ее в одной точечной оценке наиболее вероятного значения. В равенстве (20.77) в первом члене легко распознать логарифмическое правдоподобие реконструкции, встречающееся и в других автокодировщиках. Второй член пытается

сблизить приближенное апостериорное распределение $q(\mathbf{z} \mid \boldsymbol{x})$ и априорное модельное распределение $p_{\text{model}}(\boldsymbol{z})$.

В традиционных подходах к вариационному выводу и обучению q выводится с помощью алгоритма оптимизации, обычно итеративного решения уравнений неподвижной точки (раздел 19.4). Это медленно и зачастую требует умения вычислять $\mathbb{E}_{\mathbf{z}\sim q} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x})$ в замкнутой форме. Основная идея вариационного автокодировщика – обучить параметрический кодировщик (который иногда называют сетью вывода или моделью распознавания), порождающий параметры q. Если $\boldsymbol{z}$ – непрерывная переменная, то мы тогда сможем выполнить обратное распространение через примеры $\boldsymbol{z}$, выбранные из $q(\boldsymbol{z} \mid \boldsymbol{x}) = q(\boldsymbol{z}; f(\boldsymbol{x}; \boldsymbol{\theta}))$, для получения градиента по $\boldsymbol{\theta}$. В таком случае обучение состоит просто из максимизации $\mathcal{L}$ относительно параметров кодировщика и декодера. Все математические ожидания в $\mathcal{L}$ можно аппроксимировать с помощью выборки методом Монте-Карло.

Подход на основе вариационного автокодировщика элегантный, теоретически удовлетворительный и простой в реализации. Ко всему прочему, он дает отличные результаты и входит в число передовых подходов к порождающему моделированию. Главный недостаток заключается в том, что выборки из вариационных автокодировщиков, обученных на изображениях, получаются несколько размытыми. Причина этого феномена до сих пор неясна. Возможно, что размытость – свойство, внутренне присущее критерию максимального правдоподобия, по которому минимизируется $D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}})$. Как показано, на рис. 3.6. это означает, что модель назначает высокую вероятность точкам, которые встречаются в обучающем наборе, и, возможно, каким-то другим точкам. Вот эти другие точки и могут включать размытые изображения. Отчасти причина того, что модель предпочитает назначать большую массу вероятности размытым изображениям, а не каким-то другим частям пространства, состоит в том, что вариационные автокодировщики, применяемые на практике, обычно имеют нормальное распределение $p_{\text{model}}(\boldsymbol{x}, g(\boldsymbol{z}))$. Максимизация нижней границы правдоподобия такого распределения похожа на обучение традиционного автокодировщика по критерию среднеквадратической ошибки в том смысле, что склонна игнорировать признаки входа, которые занимают мало пикселей или приводят лишь к небольшому изменению яркости тех пикселей, которые занимают. Эта проблема характерна не только для VAE, но и для других порождающих моделей, которые оптимизируют логарифмическое правдоподобие или, что то же самое, $D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}})$ (см. Theis et al. (2015) и Huszar (2015)). У современных моделей VAE есть еще одна неприятная проблема – они склонны использовать лишь малое подмножество измерений $\boldsymbol{z}$, как будто кодировщик не способен преобразовать достаточно много локальных направлений в пространстве входов в пространство, где маргинальное распределение совпадает с факторизованным априорным распределением.

Инфраструктура VAE легко обобщается на разнообразные архитектуры моделей. Это важное преимущество, по сравнению с машинами Больцмана, которые требуют скрупулезно проектировать модель, чтобы избежать вычислительной неразрешимости. VAE прекрасно работают с широким семейством дифференцируемых операторов. Из особо изощренных VAE упомянем модель **глубокого рекуррентного внимательного писателя** (deep recurrent attentive writer – DRAW) (Gregor et al., 2015). В модели DRAW используются рекуррентный кодировщик и рекуррентный декодер в сочетании с механизмом внимания. Порождающий процесс в DRAW состоит из последова-

тельного посещения небольших участков изображения и выборки значений пикселей в этих точках. VAE также можно обобщить на порождение последовательностей, если определить вариационные РНС (Chung et al., 2015b), используя рекуррентные кодировщик и декодер в составе инфраструктуры VAE. Выборка из традиционных РНС включает только недетерминированные операции в пространстве выходов. Вариационные РНС обладают также возможностью рандомизации на потенциально более абстрактном уровне, улавливаемом латентными переменными VAE.

Инфраструктура VAE обобщена также на максимизацию не только традиционной вариационной нижней границы, но и на целевую функцию **автокодировщика, взвешенного по значимости** (importance-weighted autoencoder) (Burda et al., 2015):

$$\mathcal{L}_k(\boldsymbol{x}, q) = \mathbb{E}_{\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(k)}\sim q(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{1}{k}\sum_{i=1}^{k}\frac{p_{\text{model}}(\boldsymbol{x}, \boldsymbol{z}^{(i)})}{q(\boldsymbol{z}^{(i)} \mid \boldsymbol{x})}\right]. \tag{20.79}$$

При $k = 1$ эта новая целевая функция эквивалентна традиционной нижней границе $\mathcal{L}$. Но ее можно также интерпретировать как оценку истинного $\log p_{\text{model}}(\boldsymbol{x})$ с использованием выборки по значимости $\boldsymbol{z}$ из вспомогательного распределения $q(\boldsymbol{z} \mid \boldsymbol{x})$. Кроме того, она является нижней границей $\log p_{\text{model}}(\boldsymbol{x})$ и с увеличением k становится точнее.

У вариационных автокодировщиков есть много интересных связей с многопредсказательными глубокими машинами Больцмана (МП-ГМБ) и другими подходами, включающими обратное распространение по графу приближенного вывода (Goodfellow et al., 2013b; Stoyanov et al., 2011; Brakel et al., 2013). В предшествующих подходах требовалось, чтобы граф вычислений поставляла процедура вывода, например решение уравнений неподвижной точки среднего поля. Вариационный автокодировщик определен для произвольных графов вычислений, поэтому применим к более широкому классу вероятностных моделей, т. к. необязательно ограничиваться лишь моделями, для которых разрешимы уравнения неподвижной точки среднего поля. У вариационного автокодировщика есть еще одно достоинство – он увеличивает границу логарифмического правдоподобия модели, тогда как критерии для МП-ГМБ и родственных моделей в большей степени эвристические и почти не допускают вероятностной интерпретации, а лишь обеспечивают верность результатов приближенного вывода. Недостаток же VAE в том, что он обучает сеть вывода решению только одной задачи: выводу $\boldsymbol{z}$ по заданному $\boldsymbol{x}$. Более ранние методы способны выполнять приближенный вывод любого подмножества переменных по любому другому известному подмножеству, поскольку уравнения неподвижной точки среднего поля описывают, как разделяются параметры между графами вычислений для этих разных задач.

Полезное свойство вариационного автокодировщика состоит в том, что одновременное обучение параметрического кодировщика в сочетании с генераторной сетью побуждает модель обучиться предсказуемой системе координат, которую может запомнить кодировщик. Благодаря этому VAE становится отличным алгоритмом обучения многообразий. На рис. 20.6 показаны примеры многообразий низкой размерности, обученных вариационным автокодировщиком. В одном из продемонстрированных на рисунке случаев алгоритм выявил два независимых фактора вариативности в изображениях лиц: угол поворота и эмоциональное выражение.

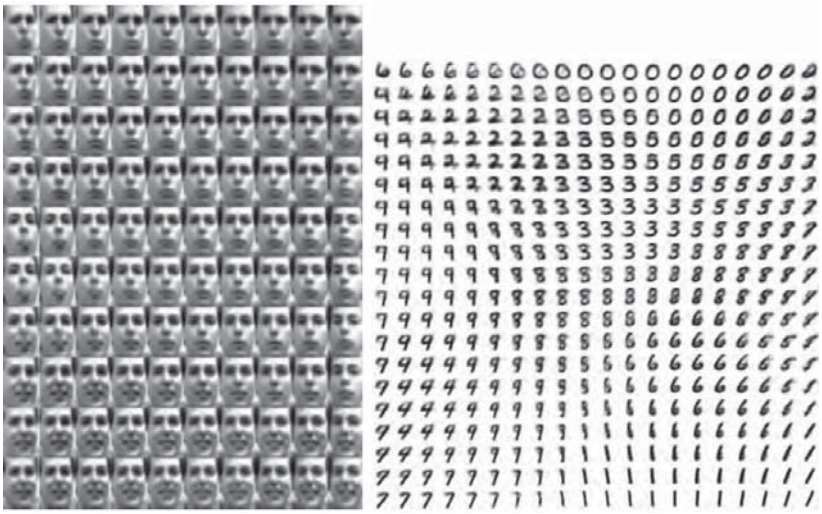

Рис. 20.6 ❖ Примеры двумерных систем координат для многообразий высокой размерности, обученных вариационным автокодировщиком (Kingma and Welling, 2014a). На странице можно нарисовать два измерения, поэтому мы можем составить некоторое представление о работе модели, обучив ее двумерному латентному коду, даже если полагаем, что истинная размерность многообразия данных гораздо выше. Показанные изображения – не примеры, взятые из обучающего набора, а изображения $\boldsymbol{x}$, фактически сгенерированные моделью $p(\boldsymbol{x} \mid \boldsymbol{z})$ путем простого изменения двумерного «кода» $\boldsymbol{z}$ (каждому изображению соответствует свой выбор «кода» $\boldsymbol{z}$ на двумерной равномерной сетке). (*Слева*) Двумерное отображение многообразия лиц Фрея. Одно выявленное измерение (по горизонтали) соответствует главным образом углу поворота лица, а другое (по вертикали) – эмоциональному выражению. (*Справа*) Двумерное отображение многообразия MNIST

20.10.4. Порождающие состязательные сети

Порождающие состязательные сети, или ПСС (Goodfellow et al., 2014c), – еще один подход к порождающему моделированию, основанный на дифференцируемых генераторных сетях. В их основе лежит теоретико-игровая ситуация, когда генераторная сеть должна состязаться с противником. Генераторная сеть непосредственно порождает примеры $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$. Ее противник, **дискриминантная сеть**, пытается отличить примеры, взятые из обучающих данных, от примеров, порожденных генератором. Дискриминатор выдает значение, возвращенное функцией $d(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$, равное вероятности того, что $\boldsymbol{x}$ – реальный обучающий пример, а не фальшивка, выбранная из модели.

Описать процесс обучения в порождающей состязательной сети проще всего как игру с нулевой суммой, в которой функция $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ определяет платеж дискриминатора. Генератор получает $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ в качестве своего платежа. В процессе обучения каждый игрок стремится максимизировать свой платеж, так что в пределе получаем

$$g^* = \arg\min_g \max_d v(g, d). \tag{20.80}$$

По умолчанию v выбирается следующим образом:

$$v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)}) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \log d(\boldsymbol{x}) + \mathbb{E}_{\boldsymbol{x}\sim p_{\text{model}}} \log(1 - d(\boldsymbol{x})). \tag{20.81}$$

Это заставляет дискриминатор пытаться обучиться правильно классифицировать примеры как настоящие или поддельные. Одновременно генератор пытается обмануть классификатор, заставив его поверить, что примеры настоящие. В пределе примеры, созданные генератором, неотличимы от настоящих данных, и дискриминатор всегда выводит $^1/_2$. После этого дискриминатор можно выбросить.

Основной побудительный мотив для проектирования ПСС состоит в том, что процесс обучения не нуждается ни в приближенном выводе, ни в аппроксимации градиента статистической суммы. Если $\max_d v(g, d)$ выпукла относительно $\boldsymbol{\theta}^{(g)}$ (как в случае, когда оптимизация производится прямо в пространстве функций плотности вероятности), то процедура гарантированно сходится и асимптотически состоятельна.

К сожалению, на практике обучение ПСС может оказаться трудным, когда g и d представлены нейронными сетями, а функция $\max_d v(g, d)$ не выпуклая. В работе Goodfellow (2014) отсутствие сходимости названо проблемой, которая может привести к недообученности ПСС. В общем случае не гарантируется, что одновременный градиентный спуск по функциям стоимости двух игроков достигнет равновесия. Рассмотрим, к примеру, функцию ценности $v(a, b) = ab$, когда один игрок контролирует a и несет потери в сумме ab, а второй контролирует b и получает $-ab$. Если мы будем моделировать каждого игрока как совершающего бесконечно малые шаги в направлении градиента, так что каждый игрок уменьшает собственные затраты за счет другого игрока, то a и b выйдут на устойчивую круговую орбиту, а не достигнут точки равновесия в начале координат. Отметим, что точки равновесия в минимаксной игре не являются локальными минимумами v. На самом деле это точки, в которых одновременно достигаются минимумы затрат обоих игроков, т. е. седловые точки v, являющиеся локальными минимумами относительно параметров первого игрока и локальными максимумами относительно параметров второго игрока. Может случиться, что оба игрока по очереди бесконечно увеличивают, а затем уменьшают v, вместо того чтобы оказаться точно в седловой точке, где ни один игрок не может уменьшить своих затрат. Неизвестно, в какой мере эта проблема несходимости затрагивает ПСС.

В работе Goodfellow (2014) предложена альтернативная формулировка платежей, при которой игра перестает иметь нулевую сумму. При этом ожидаемый градиент такой же, как при обучении с критерием максимального правдоподобия, если только дискриминатор оптимален. Поскольку обучение с критерием максимального правдоподобия сходится, при такой формулировке игры ПСС тоже должна сходиться при наличии достаточного числа примеров. Увы, на практике сходимости не наблюдается, быть может, из-за неоптимальности дискриминатора или высокой дисперсии ожидаемого градиента.

В реалистичных экспериментах наилучшей формулировкой игры ПСС является не игра с нулевой суммой и не эквивалент максимального правдоподобия, введенный в работе Goodfellow et al. (2014c) с эвристическим обоснованием. Оптимальные результаты получаются, когда генератор стремится увеличить логарифм вероятности, что дискриминатор допустит ошибку, а не уменьшить логарифм вероятности, что дискриминатор сделает правильное предсказание. В обоснование такой формулировки положено одно-единственное наблюдение: при подобной стратегии произ-

водная функции стоимости генератора по функции logit дискриминатора остается большой даже в ситуации, когда дискриминатор уверенно отклоняет все примеры генератора.

Стабилизация обучения ПСС остается открытой проблемой. По счастью, обучение ПСС хорошо работает при тщательном выборе архитектуры и гиперпараметров модели. В работе Radford et al. (2015) построена глубокая сверточная ПСС (DCGAN), показывающая отличные результаты в задачах синтеза изображений, и показано, что пространство ее латентного представления улавливает важные факторы вариативности (см. рис. 15.9). На рис. 20.7 приведены примеры изображений, порожденных генератором DCGAN.

Рис. 20.7 ❖ Изображения, сгенерированные ПСС, обученными на наборе данных LSUN. (*Слева*) Изображения спален, сгенерированные моделью DCGAN, взяты из работы Radford et al. (2015). (*Справа*) Изображения церквей, сгенерированные моделью LAPGAN, взяты из работы Denton et al. (2015)

Задачу обучения ПСС можно упростить, разбив процесс генерации на много уровней детализации. Возможно обучить условные ПСС (Mirza and Osindero, 2014), которые производят выборку из распределения $p(\boldsymbol{x} \mid \boldsymbol{y})$, а не просто из маргинального распределения $p(\boldsymbol{x})$. В работе Denton et al. (2015) показано, что последовательность условных ПСС можно обучить сначала порождать вариант изображения с очень низким разрешением, а затем постепенно добавлять детали. Эта техника называется моделью LAPGAN, поскольку для генерации изображений разного уровня детализации применяется пирамида Лапласа. Генераторы LAPGAN способны обмануть не только дискриминантные сети, но и человека; в экспериментах испытуемые определяли до 40% изображений, сгенерированных сетью, как настоящие данные. На рис. 20.7 приведены примеры изображений, созданных генератором LAPGAN.

Одна необычная особенность процедуры обучения ПСС заключается в том, что она может аппроксимировать распределения, назначающие нулевую вероятность обучающим примерам. Вместо максимизации логарифма вероятности отдельных точек генераторная сеть обучается очерчивать многообразие, точки которого чем-то напоминают обучающие точки. Парадоксально, но это означает, что модель может присвоить логарифму вероятности тестового набора значение минус бесконечность и тем не менее представлять многообразие, которое, на взгляд человека-наблюдателя,

улавливает суть задачи генерации. Это нельзя однозначно назвать ни плюсом, ни минусом. Кроме того, если мы хотим гарантировать, что генераторная сеть назначает ненулевую вероятность всем точкам, достаточно сделать так, чтобы ее последний слой прибавлял гауссов шум ко всем сгенерированным значениям. Генераторные сети, ведущие себя таким образом, производят выборку из того же распределения, которое получается, когда генераторная сеть используется для параметризации среднего значения условного нормального распределения.

Прореживание, похоже, играет важную роль в дискриминантной сети. В частности, блоки следует стохастически прореживать в процессе вычисления градиента, в направлении которого должна следовать генераторная сеть. Следование градиенту, вычисленному детерминированной версией дискриминатору с весами, поделенными на два, похоже, менее эффективно. К тому же прореживание, кажется, никогда не приносит ничего плохого.

Хотя инфраструктура ПСС предназначена для дифференцируемых генераторных сетей, похожие принципы можно использовать и для обучения моделей других видов. Например, метод **самоконтролируемого усиления** (selfsupervised boosting) применяется для обучения генераторной ОМБ, обманывающей дискриминатор на основе логистической регрессии (Welling et al., 2002).

20.10.5. Порождающие сети с сопоставлением моментов

Порождающая сеть с сопоставлением моментов (Li et al., 2015; Dziugaite et al., 2015) – еще один вид порождающей модели, основанной на дифференцируемых генераторных сетях. В отличие от VAE и ПСС, здесь не нужно комбинировать генераторную сеть с какой-то другой – ни с сетью вывода, как в VAE, ни с дискриминантной сетью, как в ПСС.

Порождающая сеть с сопоставлением моментов обучается методом **сопоставления моментов**. Его основная идея – обучить генератор так, чтобы многие статистики выборки из модели были максимально похожи на соответствующие статистики выборки из обучающего набора. В этом контексте **моментами** называются математические ожидания различных степеней случайной величины. Например, первый момент – это среднее, второй – сумма квадратов и т. д. В многомерном случае каждый элемент случайного вектора может быть возведен в разные степени, поэтому моментом может быть любая величина вида

$$\mathbb{E}_{\boldsymbol{x}} \Pi_i x_i^{n_i}, \tag{20.82}$$

где $\boldsymbol{n} = [n_1, n_2, ..., n_d]^\top$ – вектор неотрицательных целых чисел.

На первый взгляд кажется, что этот подход вычислительно неразрешим. Например, чтобы сопоставить все моменты вида $x_i x_j$, понадобится минимизировать разность между величинами, количество которых квадратично зависит от размерности $\boldsymbol{x}$. Более того, даже сопоставления всех первых и вторых моментов будет достаточно только для аппроксимации многомерного нормального распределения, улавливающего лишь линейные связи между значениями. А наши амбиции простираются на нейронные сети, которые должны улавливать сложные нелинейные связи, так что моментов потребуется куда больше. В ПСС проблемы полного перечисления всех моментов удается избежать благодаря использованию обновляемого дискриминатора, который автоматически концентрирует внимание на той статистике, которую генераторная сеть повторяет наименее эффективно.

Но вместо этого порождающую сеть с сопоставлением моментом можно обучить путем минимизации функции стоимости, называемой **максимальным средним расхождением** (maximum mean discrepancy – MMD) (Schölkopf and Smola, 2002; Gretton et al., 2012). Эта функция измеряет ошибку на первых моментах в бесконечномерном пространстве, используя неявное отображение в пространство признаков, определяемое некоторой ядерной функцией, в результате чего вычисления с бесконечномерными векторами становятся реальными. Стоимость MMD равна нулю тогда и только тогда, когда два сравниваемых распределения совпадают.

Визуально примеры, выбранные из порождающей сети с сопоставлением моментов, разочаровывают. Но, к счастью, их можно улучшить, скомбинировав генераторную сеть с автокодировщиком. Сначала автокодировщик обучается реконструировать обучающий набор, а затем его кодировщик используется для преобразования всего обучающего набора в кодовое пространство. После этого генераторная сеть обучается генерировать примеры кодов, которые можно отобразить на визуально приемлемые примеры с помощью декодера.

В отличие от ПСС, функция стоимости определена только по отношению к пакету примеров, взятых одновременно из обучающего набора и генераторной сети. Невозможно произвести обновление в виде функции только одного обучающего примера или только одного примера из генераторной сети, поскольку моменты вычисляются как эмпирическое среднее по большому числу примеров. Если размер пакета слишком мал, то MMD может дать заниженную оценку истинного различия распределений, из которых произведена выборка. Для полного устранения этой проблемы пакета конечного размера вообще недостаточно, но чем больше пакет, тем меньше занижение оценки. Если размер пакета слишком велик, то процедура обучения становится неприемлемо медленной, т. к. для вычисления одного малого шага градиента нужно обработать много примеров.

Как и в случае ПСС, обучить генераторную сеть с помощью MMD можно даже тогда, когда она назначает нулевую вероятность обучающим примерам.

20.10.6. Сверточные порождающие сети

При порождении изображений часто бывает полезно использовать генераторную сеть, включающую сверточную структуру (см., например Goodfellow et al. [2014c] или Dosovitskiy et al. [2015]). Для этого применяется «транспонированный» оператор свертки, описанный в разделе 9.5. Этот подход часто дает более реалистичные изображения да к тому же с меньшим числом параметров, чем при использовании полносвязных слоев без разделения параметров.

В сверточных сетях для решения задач распознавания поток информации течет от изображения к верхнему слою сети, где производится обобщение, часто выражаемое меткой класса. По мере распространения вверх по сети информация отбрасывается, т. к. представление изображения становится все более инвариантным к мелким преобразованиям. В генераторной сети все происходит наоборот. По мере того как изображение распространяется по сети, к нему добавляется все больше деталей, и в конечном итоге получается полноценное изображение, где присутствуют все объекты в требуемых положениях, с правильными текстурами и освещением. Основной механизм отбрасывания информации в сверточной сети распознавания – слой пулинга. Но генераторная сеть должна добавлять информацию. Мы не можем включить в генераторную сеть слой инверсного пулинга, поскольку функции пулинга

в большинстве своем необратимы. Есть более простая операция – увеличить пространственный размер представления. В работе Dosovitskiy et al. (2015) применен «антипулинг», который, похоже, дает удовлетворительные результаты. Этот слой соответствует обращению операции max-пулинга при некоторых упрощающих условиях. Во-первых, шаг операции max-пулинга должен совпадать с шириной области пулинга. Во-вторых, предполагается, что максимальный элемент в каждой области пулинга расположен в левом верхнем углу. Наконец, предполагается, что все немаксимальные элементы в каждой области пулинга равны нулю. Это очень сильные и нереалистичные предположения, но они позволяют обратить операцию max-пулинга. Операция антипулинга выделяет память для нулевого тензора, а затем копирует каждое входное значение с пространственной координатой i в выходное значение с пространственной координатой $i \times k$. Целое число k определяет размер области пулинга. Хотя предположения, на которых базируется оператор антипулинга, нереалистичны, последующие слои могут обучиться компенсировать необычный выход, поэтому примеры, генерируемые моделью в целом, визуально приятны.

20.10.7. Авторегрессивные сети

Авторегрессивные сети – это ориентированные вероятностные модели без латентных случайных переменных. Условные распределения вероятности в этих моделях представлены нейронными сетями (иногда очень простыми типа логистической регрессии). Модели соответствует полный граф. Совместное распределение наблюдаемых переменных в таких моделях с помощью цепного правила вероятностей разлагается в произведение условных распределений вида $P(x_d \mid x_{d-1}, ..., x_1)$. Такие модели под названием «**полностью видимые байесовские сети**» (ПВБС, англ. FVBN) успешно использовались в различных формах, сначала с логистической регрессией в качестве каждого условного распределения (Frey, 1998), а затем с нейронными сетями со скрытыми блоками (Bengio and Bengio, 2000b; Larochelle and Murray, 2011). В некоторых вариантах авторегрессивных сетей, например NADE (Larochelle and Murray, 2011), описанной в разделе 20.10.10, можно реализовать некий вид разделения параметров, что дает как статистический (меньше уникальных параметров), так и вычислительный (меньше объем вычислений) выигрыш. Это еще один пример повторяющегося в глубоком обучении мотива повторного использования признаков.

20.10.8. Линейные авторегрессивные сети

В простейшей форме авторегрессивных сетей нет ни скрытых блоков, ни разделения параметров. Каждое условное распределение $P(x_i \mid x_{i-1}, ..., x_1)$ параметризуется как линейная модель (линейная регрессия в случае вещественных данных, логистическая регрессия в случае бинарных данных и softmax-регрессия в случае дискретных данных). Эта модель впервые была предложена в работе Frey (1998), в ней $O(d^2)$ параметров, где d – количество переменных в модели. Она показана на рис. 20.8.

Если переменные непрерывны, то линейная авторегрессивная сеть – просто еще одна формулировка многомерного нормального распределения, улавливающего линейные попарные взаимодействия между наблюдаемыми величинами.

По сути своей линейные авторегрессивные сети – это обобщение методов линейной классификации на порождающее моделирование. Поэтому у них такие же плюсы и минусы, как у линейных классификаторов. Их точно так же можно обучать с помощью выпуклых функций потерь, и иногда они допускают решение в замкнутой

форме (как в случае нормального распределения). Как и линейный классификатор, сама по себе модель не предлагает никакого способа увеличения емкости, так что для этой цели следует использовать такие приемы, как расширение базиса входа или трюк с ядром.

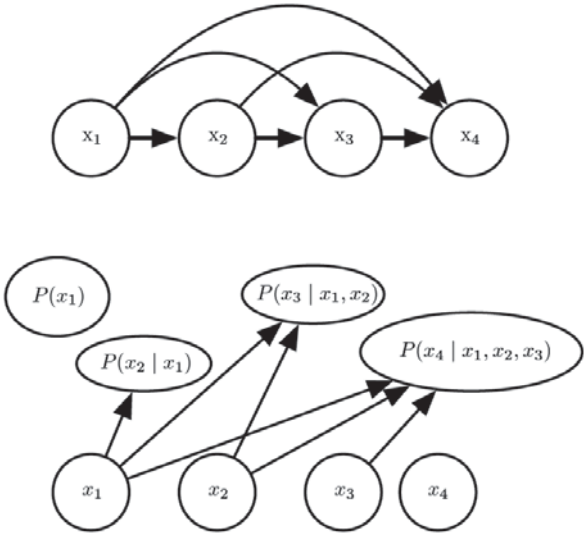

Рис. 20.8 ❖ Полностью видимая байесовская сеть предсказывает *i*-ю переменную по $i - 1$ предыдущим. (*Сверху*) Ориентированная графическая модель ПВБС. (*Снизу*) Граф вычислений для логистической ПВБС, в которой каждое предсказание делается линейным предиктором

20.10.9. Нейронные авторегрессивные сети

Нейронным авторегрессивным сетям (Bengio and Bengio, 2000a,b) соответствует такая же направленная слева направо графическая модель, как логистическим авторегрессивным сетям (рис. 20.8), но внутри этой графической структуры используется другая параметризация условных распределений. Она мощнее в том смысле, что емкость модели можно увеличивать как угодно и, значит, аппроксимировать любое совместное распределение. Кроме того, новая параметризация улучшает обобщаемость благодаря принципу разделения параметров и признаков, присущему глубокому обучению вообще. Модель была предложена, чтобы уйти от проклятия размерности, имеющего место в традиционных табличных графических моделях с такой же структурой, как на рис. 20.8. В табличных дискретных вероятностных моделях каждое условное распределение представлено таблицей вероятностей, в которой каждой возможной конфигурации участвующих переменных соответствуют одна ячейка и один параметр. Использование вместо этого нейронной сети дает два преимущества:

1) параметризация каждого $P(x_i \mid x_{i-1}, ..., x_1)$ нейронной сетью с $(i - 1) \times k$ входами и k выходами (если переменные дискретны и принимают k значений, представленных унитарным кодом) позволяет оценить условную вероятность, не требуя экспоненциального числа параметров (и примеров), но при этом еще и способна уловить зависимости высшего порядка между случайными величинами;

2) вместо отдельной нейронной сети для предсказания каждого x_i направленные *слева направо* связи (рис. 20.9) позволяют объединить все сети в одну. Иначе говоря, это означает, что признаки скрытого слоя, вычисленные для предсказания x_i, можно повторно использовать для предсказания x_{i+k} ($k > 0$). Таким образом, скрытые блоки организованы в *группы*, обладающие тем свойством, что все блоки i-й группы зависят только от входных значений x_1, ..., x_i. Параметры, используемые для вычисления этих скрытых блоков, совместно оптимизируются с целью улучшить предсказание всех переменных в последовательности. Это пример *принципа повторного использования*, который пронизывает все глубокое обучения – от архитектур рекуррентных и сверточных сетей до многозадачного обучения и переноса обучения.

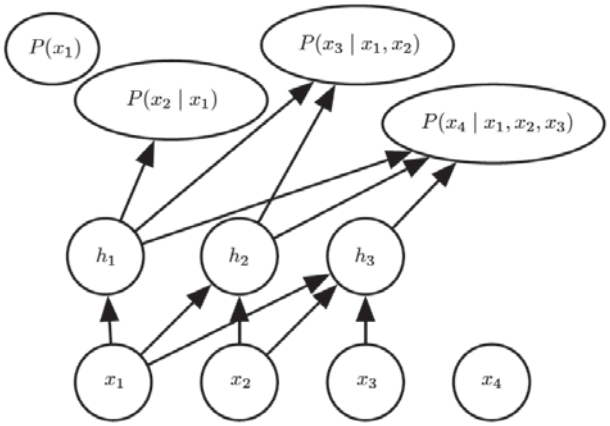

Рис. 20.9 ❖ Нейронная авторегрессивная сеть предсказывает i-ю переменную x_i по $i-1$ предыдущим, но параметризована так, что признаки (группы скрытых блоков, обозначенные h_i), являющиеся функциями от x_1, ..., x_i, можно повторно использовать для предсказания всех последующих переменных x_{i+1}, x_{i+2}, ..., x_d

Каждое $P(x_i \mid x_{i-1}, ..., x_1)$ может представлять условное распределение, если выходы нейронной сети будут предсказывать *параметры* условного распределения x_i, как обсуждалось в разделе 6.2.1.1. Хотя первоначально авторегрессивные сети работали с чисто дискретными многомерными данными (с сигмоидным выходным блоком для случайных величин с распределением Бернулли или softmax-блоком для величин с категориальным распределением), они естественно обобщаются на непрерывные величины или совместные распределения, включающие как дискретные, так и непрерывные величины.

20.10.10. NADE

Нейронный авторегрессивный оцениватель плотности (neural auto-regressive density estimator – NADE) – недавно появившаяся и уже ставшая очень успешной форма нейронной авторегрессивной сети (Larochelle and Murray, 2011). Связность в ней такая же, как в оригинальной нейронной авторегрессивной сети из работы Bengio and Bengio (2000b), но введена дополнительная схема разделения параметров, показанная на рис. 20.10. Параметры скрытых блоков из разных групп j разделяются.

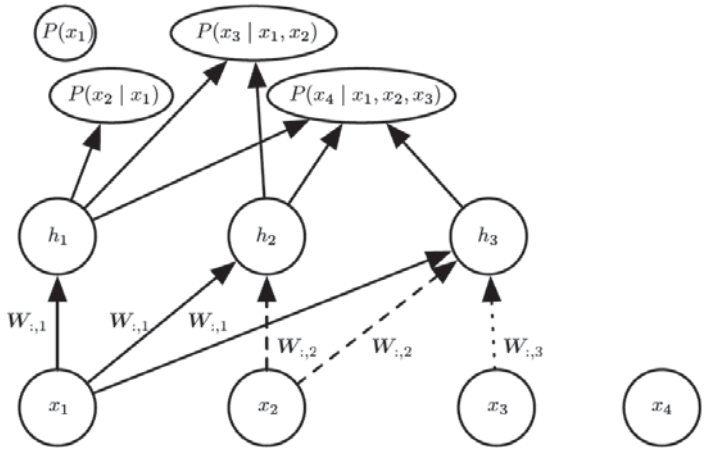

Рис. 20.10 ❖ Нейронный авторегрессивный оцениватель плотности (NADE). Скрытые блоки организованы в группы $\boldsymbol{h}^{(j)}$, так что только входы x_1, ..., x_i участвуют в вычислении $\boldsymbol{h}^{(i)}$ и предсказании $P(x_j \mid x_{j-1}, ..., x_1)$ для $j > i$. NADE отличается от более ранних нейронных авторегрессивных сетей использованием специальной схемы разделения весов: значение $W'_{j,k,i} = W_{k,i}$ одинаково (на рисунке это обозначено одинаковым стилем линий для всех экземпляров повторяющегося веса) для всех весов связей, идущих из x_i в k-ый блок любой группы $j \geq i$. Напомним, что вектор $(W_{1,i}, W_{2,i}, ..., W_{n,i})$ обозначается $\boldsymbol{W}_{:,i}$

Веса $W'_{j,k,i}$ связей между i-м входом x_i и k-ым элементом j-ой группы скрытых блоков $h_k^{(j)} (j \geq i)$ разделяются между группами:

$$W'_{j,k,i} = W_{k,i}. \tag{20.83}$$

Остальные веса (для $j < i$) равны нулю.

В работе Larochelle and Murray (2011) выбрана эта схема разделения, так что прямое распространение в NADE чем-то напоминает вычисления, которые производятся в выводе среднего поля для восполнения отсутствующих значений в ОМБ. Этот вывод среднего поля соответствует выполнению рекуррентной сети с разделяемыми весами, и первый шаг вывода такой же, как в NADE. Единственное отличие состоит в том, что в NADE веса связей скрытых блоков с выходными параметризуются независимо от весов связей входных блоков со скрытыми. В ОМБ матрица весов связей скрытые–выходные является транспонированной к матрице весов связей входные–скрытые. Архитектуру NADE можно обобщить, так чтобы она имитировала не один, а k временных шагов рекуррентного вывода среднего поля. Этот подход называется NADE-k (Raiko et al., 2014).

Как уже отмечалось, авторегрессивные сети можно распространить на обработку непрерывных данных. Особенно эффективным и общим способом параметризации непрерывной плотности является смесь гауссовых распределений с весами α_i (коэффициент или априорная вероятность i-й компоненты), условными средними μ_i и условными дисперсиями σ_i^2. В модели RNADE (Uria et al., 2013) такая параметризация используется для обобщения NADE на вещественные значения. Как и в других сетях со смесовой плотностью, параметры этого распределения являются выходами сети, причем веро-

ятности весов компонент порождаются softmax-блоком, а дисперсии положительны. Метод стохастического градиентного спуска может оказаться численно неустойчивым из-за взаимодействий между условными средними и условными дисперсиями. Для преодоления этой трудности в работе Uria et al. (2013) на этапе обратного распространения используется псевдоградиент, заменяющий градиент по среднему.

В еще одном очень интересном обобщении нейронных авторегрессивных архитектур удалось избавиться от необходимости выбирать произвольный порядок наблюдаемых переменных (Murray and Larochelle, 2014). Идея авторегрессивной сети – обучить сеть справляться с любым порядком за счет того, что порядок выбирается случайно, а скрытым блокам передается информация о том, какие входные данные наблюдались (находятся справа от вертикальной черты в формуле условной вероятности), а какие считаются отсутствующими и подлежат предсказанию (находятся слева от вертикальной черты). Это хорошо, потому что позволяет весьма эффективно использовать обученную авторегрессивную сеть для *решения любой проблемы вывода* (т. е. предсказывать или производить выборку из распределения вероятности любого подмножества переменных при условии любого другого подмножества). Наконец, поскольку возможно много порядков переменных ($n!$ для n переменных) и каждый порядок o переменных дает различные вероятности $p(\mathbf{x} \mid o)$, мы можем образовать ансамбль моделей для нескольких значений o:

$$p_{\text{ensemble}}(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} p(\mathbf{x} \mid o^{(i)}). \tag{20.84}$$

Этот ансамбль моделей обычно лучше обообщается и назначает тестовому набору более высокую вероятность, чем отдельная модель, определенная одним упорядочением.

В той же работе авторы предлагают глубокие варианты этой архитектуры, но, к сожалению, вычисления при этом сразу же становятся такими же дорогостоящими, как в оригинальной нейронной авторегрессивной сети (Bengio and Bengio, 2000b). Первый и выходной слои все еще можно вычислить за $O(nh)$ операций умножения-сложения, как в стандартном алгоритме NADE, где h – число скрытых блоков (размер групп h_i на рис. 20.10 и 20.9), тогда как в работе Bengio and Bengio (2000b) таких операций $O(n^2h)$. Но для других скрытых слоев сложность вычислений имеет порядок $O(n^2h^2)$, если каждая «предыдущая» группа в слое l участвует в предсказании «следующей» группы в слое $l + 1$, а n – число групп из h скрытых блоков в каждом слое. Если предположить, что i-я группа в слое $l + 1$ зависит только от i-й группы в слое l, как в работе Murray and Larochelle (2014), то сложность уменьшится до $O(nh^2)$, но это все равно в h раз хуже, чем в стандартном NADE.

20.11. Выборка из автокодировщиков

В главе 14 мы видели, что многие виды автокодировщиков обучаются распределению данных. Существуют тесные связи между сопоставлением рейтингов, шумоподавляющими автокодировщиками и сжимающими автокодировщиками. Они показывают, что некоторые автокодировщики каким-то образом обучаются распределению данных. Но мы еще не видели, как производить выборку из таких моделей.

Некоторые автокодировщики, например вариационные, явно представляют распределение вероятности и допускают прямую предковую выборку. Для большинства других необходима выборка MCMC-методами.

Сжимающие автокодировщики предназначены для восстановления оценки касательной плоскости к многообразию данных. Это означает, что повторяющееся кодирование и декодирование с привнесенным шумом индуцирует случайное блуждание по поверхности многообразия (Rifai et al., 2012; Mesnil et al., 2012). Эта техника диффузии на многообразии является разновидностью марковской цепи.

Существует также более общая марковская цепь, способная производить выборку из любого шумоподавляющего автокодировщика.

20.11.1. Марковская цепь, ассоциированная с произвольным шумоподавляющим автокодировщиком

В обсуждении выше остался открытым вопрос о том, какой шум привносить и где взять марковскую цепь, которая будет генерировать примеры из распределения, оцененного автокодировщиком. В работе Bengio et al. (2013c) показано, как построить такую марковскую цепь для **обобщенных шумоподавляющих автокодировщиков**. Такие автокодировщики задаются шумоподавляющим распределением для выборки из оценки чистого входа при известном зашумленном входе.

Каждый шаг марковской цепи, генерирующей примеры из оценки распределения, состоит из следующих подшагов, показанных на рис. 20.11:

1. Начав с предыдущего состояния $\boldsymbol{x}$, привнести искажающий шум, выбирая $\tilde{\boldsymbol{x}}$ из $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.
2. Закодировать $\tilde{\boldsymbol{x}}$ в $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$.
3. Декодировать $\boldsymbol{h}$ и получить параметры $\boldsymbol{\omega} = g(\boldsymbol{h})$ распределения $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.
4. Выбрать следующее состояние $\boldsymbol{x}$ из $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.

В работе Bengio et al. (2014) показано, что если автокодировщик $p(\mathbf{x} \mid \tilde{\mathbf{x}})$ дает состоятельную оценку соответствующего условного распределения, то стационарное распределение вышеуказанной марковской цепи дает состоятельную оценку (хотя и неявную) порождающего данные распределения $\mathbf{x}$.

20.11.2. Фиксация и условная выборка

Как и машины Больцмана, шумоподавляющие автокодировщики и их обобщения (например, описанные ниже порождающие стохастические сети) можно использовать для выборки из условного распределения $p(\mathbf{x}_f \mid \mathbf{x}_o)$, просто зафиксировав *наблюдаемые* блоки $\mathbf{x}_o$ и производя повторную выборку только *свободных* блоков $\mathbf{x}_f$ при условии $\mathbf{x}_o$ и выбранных латентных переменных (если таковые существуют). Например, МП-ГМБ можно интерпретировать как вариант шумоподавляющего автокодировщика и использовать для выборки отсутствующих входов. Позднее порождающие стохастические сети обобщили некоторые идеи, присутствующие в МП-ГМБ для выполнения той же операции (Bengio et al., 2014). В работе Alain et al. (2015) отмечено условие, пропущенное в предложении 1 из работы Bengio et al. (2014), а именно что оператор перехода (стохастическое отображение текущего состояния цепи в следующее) должен обладать свойством **детального баланса**, означающим, что марковская цепь, находящаяся в состоянии равновесия, будет оставаться в нем вне зависимости от того, выполняется оператор перехода в прямом или обратном направлении.

На рис. 20.12 показан эксперимент, состоящий в фиксации половины пикселей (правая часть изображения) и выполнении марковской цепи в другой половине.

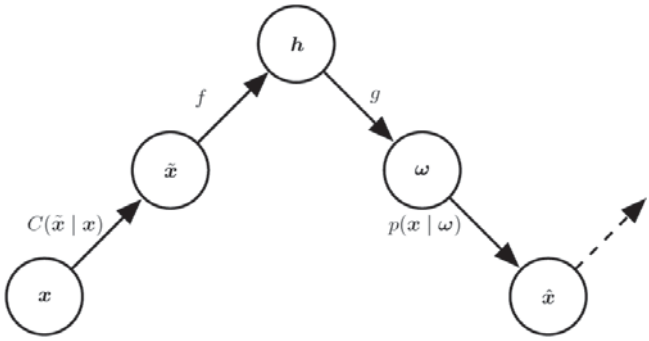

Рис. 20.11 ❖ Шаг марковской цепи, ассоциированной с обученным шумоподавляющим автокодировщиком, которая генерирует примеры из вероятностной модели, неявно обученной с критерием шумоподавляющего логарифмического правдоподобия. Каждый шаг состоит из: (*a*) привнесения шума в состояние $\boldsymbol{x}$ с помощью искажающего процесса $\boldsymbol{C}$, который дает $\tilde{\boldsymbol{x}}$; (*b*) кодирования $\tilde{\boldsymbol{x}}$ с помощью функции f и получения $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$; (*c*) декодирования результата с помощью функции g с получением параметров $\boldsymbol{\omega}$ для реконструкции распределения и (*d*) выборки нового состояния из реконструированного распределения $p(\mathbf{x} \mid \boldsymbol{\omega} = g(f(\tilde{\boldsymbol{x}})))$ при известном $\boldsymbol{\omega}$. В типичном случае среднеквадратической ошибки реконструкции $g(\boldsymbol{h}) = \hat{\boldsymbol{x}}$, которая оценивает $\mathbb{E}[\boldsymbol{x} \mid \tilde{\boldsymbol{x}}]$, искажение сводится к прибавлению гауссова шума, а выборка из $p(\mathbf{x} \mid \boldsymbol{\omega})$ – к повторному прибавлению гауссова шума к реконструкции $\hat{\boldsymbol{x}}$. Второе прибавление шума должно соответствовать среднеквадратическим ошибкам реконструкции, а привнесенный шум – это гиперпараметр, управляющий скоростью приработки и степенью сглаживания эмпирического распределения оценивателем (Vincent, 2011). В приведенном примере только условные распределения C и p – стохастические шаги (вычисление f и g детерминировано), хотя шум можно привносить и внутри автокодировщика, как в порождающих стохастических сетях (Bengio et al., 2014)

20.11.3. Возвратная процедура обучения

Возвратная (walk-back) процедура обучения была предложена в работе Bengio et al. (2013c) как способ ускорить сходимость порождающего обучения шумоподавляющих автокодировщиков. Вместо выполнения одного шага реконструкции кодирование-декодирование в этой процедуре попеременно выполняется несколько стохастических шагов кодирования-декодирования (как в порождающей марковской цепи). Процедура начинается с обучающего примера (как в алгоритме сопоставительного расхождения, описанном в разделе 18.2) и штрафует последние вероятностные реконструкции (или все реконструкции на пути).

Обучение с k шагами эквивалентно (в том смысле, что достигается то же самое стационарное распределение) обучению с одним шагом, но обладает практическим преимуществом: паразитные моды вдали от данных устраняются эффективнее.

Рис. 20.12 ❖ Фиксация правой половины изображения и выполнение марковской цепи путем повторной выборки только из левой половины на каждом шаге. Примеры взяты из порождающей стохастической сети, обученной реконструировать цифры из набора MNIST на каждом шаге с помощью возвратной процедуры

20.12. Порождающие стохастические сети

Порождающие стохастические сети (generative stochastic networks – GSN) (Bengio et al., 2014) – это обобщение шумоподавляющих автокодировщиков. Они включают латентные переменные $\mathbf{h}$ в порождающей марковской цепи, помимо видимых переменных (обычно обозначаемых $\mathbf{x}$).

GSN параметризуется двумя условными распределениями вероятности, описывающими один шаг марковской цепи.

1. $p(\mathbf{x}^{(k)} \mid \mathbf{h}^{(k)})$ говорит, как сгенерировать следующую видимую переменную, если известно текущее латентное состояние. Такое «распределение реконструкции» встречается также в шумоподавляющих автокодировщиках, ОМБ, ГСД и ГМБ.
2. $p(\mathbf{h}^{(k)} \mid \mathbf{h}^{(k-1)}, \mathbf{x}^{(k-1)})$ говорит, как обновить латентную переменную состояния, если известны предыдущее латентное состояние и видимая переменная.

Шумоподавляющие автокодировщики и GSN отличаются от классических вероятностных моделей (ориентированных и неориентированных) тем, что параметризуют сам порождающий процесс, а не математическую спецификацию совместного распределения видимых и латентных переменных. Последнее же (если существует) определяется неявно как стационарное распределение порождающей марковской цепи. Условия существования стационарного распределения довольно мягкие – такие же, как в стандартных MCMC-методах (см. раздел 17.3). Это необходимые условия приработки цепи, но существуют такие переходные распределения (например, детерминированные), для которых они нарушаются.

Можно представить себе различные критерии обучения GSN. В работе Bengio et al. (2014) предложено просто использовать логарифм вероятности реконструкции для видимых блоков, как в шумоподавляющих автокодировщиках. Для этого нужно зафиксировать $\mathbf{x}^{(0)} = \boldsymbol{x}$, наблюдаемому примеру, и максимизировать вероятность порождения $\boldsymbol{x}$ на протяжении какого-то количества последующих временных шагов, т. е. максимизировать $\log p(\mathbf{x}^{(k)} = \boldsymbol{x} \mid \mathbf{h}^{(k)})$, где $\mathbf{h}^{(k)}$ выбирается из цепи при условии $\mathbf{x}^{(0)} = \boldsymbol{x}$. Чтобы оценить градиент $\log p(\mathbf{x}^{(k)} = \boldsymbol{x} \mid \mathbf{h}^{(k)})$ по другим частям модели, в работе Bengio et al. (2014) используется трюк с перепараметризацией, описанный в разделе 20.9, а для улучшения сходимости – возвратная процедура обучения из раздела 20.11.3.

20.12.1. Дискриминантные GSN

Оригинальная формулировка GSN (Bengio et al., 2014) предназначалась для обучения без учителя и неявного моделирования $p(\mathbf{x})$ для наблюдаемых данных $\mathbf{x}$, но подход можно модифицировать для оптимизации $p(\mathbf{y} \mid \boldsymbol{x})$.

Например, в работе Zhou and Troyanskaya (2014) GSN обобщены в этом направлении, для чего производилось обратное распространение логарифма вероятности реконструкции только выходных переменных, а входные оставались фиксированными. Авторы успешно применили эту идею к моделированию последовательностей (вторичной структуры белков) и ввели одномерную сверточную структуру в оператор перехода марковской цепи. Важно не забывать, что на каждом шаге марковской цепи порождается новая последовательность для каждого слоя и что эта последовательность является входом для вычисления значений в других слоях (скажем, выше и ниже данного) на следующем шаге.

Таким образом, марковская цепь на самом деле построена только над выходной переменной (и ассоциированными скрытыми слоями верхних уровней), а входная последовательность служит только для обусловливания этой цепи. При этом обратное распространение позволяет модели обучиться, как входная последовательность может обусловить выходное распределение, неявно представленное марковской цепью. Следовательно, в этом случае GSN используется в контексте структурного выхода.

В работе Zöhrer and Pernkopf (2014) предложена гибридная модель, объединяющая в себе обучение с учителем (как в описанной выше работе) и без учителя (как в оригинальной работе по GSN). Для этого просто складываются (с разными весами) стоимости обучения с учителем и без учителя, т. е. логарифмы вероятности реконструкции $\mathbf{y}$ и $\mathbf{x}$ соответственно. Такой гибридный критерий ранее предлагался для ОМБ в работе Larochelle and Bengio (2008), где авторам удалось с помощью этой схемы улучшить качество классификации.

20.13. Другие схемы порождения

В описанных выше методах для порождения примеров использовалась либо МСМС-выборка, либо предковая выборка, либо та или иная их комбинация. Хотя это наиболее популярные подходы к порождающему моделированию, они ни в коем случае не единственные.

В работе Sohl-Dickstein et al. (2015) для обучения порождающей модели разработана схема **инверсии диффузии**, основанная на неравновесной термодинамике. Основная идея состоит в том, что у распределений вероятности, из которых мы хотим производить выборку, имеется структура. Эта структура может постепенно разрушаться

процессом диффузии, который изменяет распределение в сторону большей энтропии. Для формирования порождающей модели мы можем обратить этот процесс, обучив модель, которая постепенно восстанавливает структуру бесструктурного распределения. Итеративно применяя процесс сближения распределения с целевым, мы можем подойти к целевому распределению. Этот подход напоминает MCMC-методы в том смысле, что для порождения выборки нужно много итераций. Однако модель определена как распределение вероятности, порождаемое последним шагом цепи. В этом смысле итеративная процедура не индуцирует никакой аппроксимации. Описанный подход также очень близок к порождающей интерпретации шумоподавляющего автокодировщика (раздел 20.11.1). Как и шумоподавляющий автокодировщик, инверсия диффузии обучает оператор перехода, который пытается вероятностно компенсировать эффект сложения с каким-то шумом. Разница в том, что инверсия диффузии требует отменить только один шаг диффузионного процесса, а не пройти назад по всему пути к чистым данным. Тем самым устраняется дилемма, присущая обычной для шумоподавляющих автокодировщиков целевой функции логарифма вероятности реконструкции: при малом уровне шума обучаемый видит только конфигурации рядом с примерами данных, а при большом задача становится почти неразрешимой (поскольку шумоподавляющее распределение становится слишком сложным и многомодальным). В случае же целевой функции инверсии диффузии обучаемый может более точно восстановить форму функции плотности в окрестности данных, а заодно избавиться от паразитных мод вдали от данных.

Еще один подход к генерации примеров – **приближенные байесовские вычисления** (approximate Bayesian computation – ABC) (Rubin et al., 1984). В этом случае примеры отклоняются или модифицируются, так чтобы моменты выбранных функций примеров совпадали с моментами желаемого распределения. В этой идее используются моменты примеров, как в алгоритме сопоставления моментов, но есть и различия, поскольку здесь производится модификация самих примеров, а не обучение модели автоматически выдавать примеры с правильными моментами. В работе Bachman and Precup (2015) показано, как идеи ABC можно использовать в контексте глубокого обучения для формирования траекторий MCMC в порождающих стохастических сетях.

Мы полагаем, что своего открытия ждет много других подходов к порождающему моделированию.

20.14. Оценивание порождающих моделей

Исследователям, изучающим порождающие модели, часто бывает необходимо сравнить две модели, обычно чтобы продемонстрировать, что новая модель лучше улавливает некоторое распределение, чем предыдущие.

Это может оказаться непростой задачей. Нередко точно вычислить логарифм вероятности данных в модели невозможно, приходится довольствоваться только аппроксимацией. В таких случаях важно отчетливо понимать и сообщать аудитории, что именно измеряется. Предположим, к примеру, что мы вычисляем стохастическую оценку логарифмического правдоподобия модели A и детерминированную нижнюю границу логарифмического правдоподобия модели B. Если модель A получила больше баллов, чем модель B, то какая из них лучше? Если нас интересует, какая модель дает лучшее внутреннее представление распределения, то ответить на этот вопрос нельзя, если только нет какого-то способа узнать, насколько точна нижняя граница

для модели B. Если же нам интересно практическое использование модели, например для обнаружения аномалий, то будет справедливо судить модели на основе критерия, относящегося к конкретной задаче, например по результатам ранжирования тестовых примеров с помощью таких критериев, как точность и полнота.

Еще одна тонкость оценивания порождающих моделей состоит в том, что выработка критериев оценки сама по себе представляет трудную научную задачу. Может оказаться очень сложно установить, что сравнение моделей производится справедливо. Предположим, к примеру, что мы используем метод AIS для получения оценки $\log Z$ с целью вычислить $\log \tilde{p}(\boldsymbol{x}) - \log Z$ для новой придуманной нами модели. Вычислительно экономная реализация AIS может не найти несколько мод модельного распределения и дать заниженную оценку Z, что приведет к завышенной оценке $\log p(\boldsymbol{x})$. Таким образом, трудно сказать, что стало причиной высокой оценки правдоподобия: хорошая модель или плохая реализация AIS.

В других разделах машинного обучения обычно допускается некоторая вариативность на этапе предобработки данных. Например, при сравнении верности алгоритмов распознавания объектов обычно разрешается производить предобработку входных изображений немного по-разному в соответствии с требованиями, предъявляемыми каждым алгоритмом. Порождающее моделирование устроено иначе – любые изменения в способе предобработки, пусть даже совсем незначительные и незаметные, абсолютно недопустимы. Всякое изменение входных данных изменяет подлежащее выявлению распределение и кардинальным образом меняет задачу. Например, умножение входных данных на 0.1 искусственно повышает правдоподобие в 10 раз.

Проблемы предобработки часто возникают при проверке порождающих моделей на эталонном наборе данных MNIST, одном из самых популярных для тестирования таких моделей. В этом наборе есть только полутоновые изображения. В одних моделях изображения из MNIST рассматриваются как точки в вещественном векторном пространстве, в других – как бинарные изображения. А в третьих полутоновые значения яркости трактуются как вероятности бинарных примеров. Важно сравнивать вещественные модели только с другими вещественными моделями, а бинарные – только с другими бинарными. В противном случае правдоподобие будет измеряться в разных пространствах. Для бинарных моделей логарифмическое правдоподобие не может быть больше нуля, тогда как в вещественных оно не ограничено сверху, будучи результатом измерения плотности. При сравнении бинарных моделей важно, чтобы применялся один и тот же вид бинаризации. Например, для сопоставления полутоновому пикселю значения 0 или 1 мы можем сравнить его с порогом 0.5 или произвести случайную выборку, в которой вероятность получить 1 определяется яркостью пикселя. Если используется случайная бинаризация, то мы можем бинаризовать весь набор данных сразу или выбрать разные случайные примеры для каждого шага обучения, а затем произвести множественную выборку для оценивания. Все три схемы дадут совершенно разные значения правдоподобия, а при сравнении разных моделей важно, чтобы использовалась одна и та же схема бинаризации для обучения и оценивания. На самом деле при выполнении единственного шага случайной бинаризации обычно создается общий файл, содержащий ее результаты, чтобы исключить расхождения из-за различных исходов шага бинаризации.

Поскольку способность порождать реалистичные примеры из распределения данных – одна из целей порождающей модели, на практике такие модели часто оценивают, визуально исследуя примеры. Лучше, когда это делает не сам исследователь,

а участник эксперимента, которому неизвестен источник происхождения примеров (Denton et al., 2015). К сожалению, бывает так, что очень плохая вероятностная модель порождает очень хорошие примеры. Общепринятый способ проверить, что модель не просто копирует какие-то обучающие примеры, иллюстрируется на рис. 16.1. Идея в том, чтобы для некоторых порожденных примеров показать их ближайших соседей в обучающем наборе согласно евклидову расстоянию в пространстве $\boldsymbol{x}$. Эта проверка направлена на то, чтобы выявить случай, когда модель переобучена и просто воспроизводит обучающие примеры. Может даже случиться, что модель одновременно переобучена и недообучена и тем не менее порождает примеры, которые по отдельности выглядят отлично. Представьте себе порождающую модель, обученную на изображениях собак и кошек, которая просто научилась воспроизводить изображения собак. Очевидно, что такая модель переобучена, поскольку она не порождает изображения, которых не было в обучающем наборе, но она также недообучена, т. к. назначает нулевую вероятность обучающим изображениям кошек. Тем не менее человек сочтет, что каждое отдельное изображение собаки высокого качества. Это простой пример – наблюдатель, просмотревший много примеров, заметит отсутствие кошек. В более реалистичных условиях порождающая модель, обученная на данных с десятками тысяч мод, может проигнорировать небольшое число мод, и человеку будет нелегко заметить, что какая-то вариация отсутствует.

Поскольку визуальное качество примеров – ненадежный путеводитель, мы часто оцениваем также логарифмическое правдоподобие, которое модель назначает данным, если это вычислительно осуществимо. К сожалению, в некоторых случаях правдоподобие не измеряет интересующих нас атрибутов модели. Например, на наборе данных MNIST вещественная модель может получить произвольно высокое правдоподобие, если назначит произвольно низкую дисперсию пикселям фона, которые никогда не изменяются. Модели и алгоритмы, которые обнаруживают такие постоянные признаки, могут быть вознаграждены не по заслугам, потому что особой пользы в этом свойстве нет. Потенциальная возможность достичь стоимости, стремящейся к минус бесконечности, существует для любого вида задач с критерием максимального правдоподобия с вещественными значениями, но особенно от этого страдают порождающие модели, оцениваемые на наборе MNIST, потому что количество тривиально предсказываемых выходных значений очень велико. Поэтому возникает настоятельная необходимость в разработке других способов оценивания порождающих моделей.

В работе Theis et al. (2015) приведен обзор многих проблем, возникающих при оценивании порождающих моделей, включающий и описанные выше соображения. Авторы подчеркивают, что порождающие модели применяются для самых разных целей и что выбор метрики должен соответствовать назначению модели. Так, одни порождающие модели лучше назначают высокую вероятность самым реалистичным точкам, тогда как другие преуспевают в редком назначении высокой вероятности нереалистичным точкам. Такие различия могут быть связаны с тем, проектировалась ли модель для минимизации $D_{\mathrm{KL}}(p_{\mathrm{data}} \| p_{\mathrm{model}})$ или $D_{\mathrm{KL}}(p_{\mathrm{model}} \| p_{\mathrm{data}})$, как показано на рис. 3.6. К сожалению, даже если ограничиться использованием только метрик, отвечающих задаче, у всех известных в настоящее время метрик имеются серьезные недостатки. Поэтому одно из самых важных направлений исследований в области порождающего моделирования – не улучшение самих моделей, а проектирование новых методов измерения успеха.

20.15. Заключение

Обучение порождающих моделей со скрытыми блоками – эффективный способ научить модель понимать мир, представленный обучающими данными. Обучившись распределению $p_{\text{model}}(\boldsymbol{x})$ и представлению $p_{\text{model}}(\boldsymbol{h} \mid \boldsymbol{x})$, порождающая модель может давать ответы на многие вопросы о связях между входными переменными в $\boldsymbol{x}$ и предлагать другие способы представления $\boldsymbol{x}$ путем вычисления математических ожиданий $\boldsymbol{h}$ на разных слоях иерархии. Порождающие модели выполняют обещание снабдить системы ИИ инфраструктурой для понимания многообразных интуитивных концепций и наделить их возможностью рассуждать об этих концепциях в условиях неопределенности. Мы надеемся, что читатели этой книги придумают новые способы повысить эффективность этих подходов и пойдут дальше по пути понимания принципов, лежащих в основе обучения и интеллекта.

Список литературы

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
2. Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. Cognitive Science, 9, 147–169.
3. Alain, G. and Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In ICLR'2013, arXiv:1211.4246 .
4. Alain, G., Bengio, Y., Yao, L., Éric Thibodeau-Laufer, Yosinski, J., and Vincent, P. (2015). GSNs: Generative stochastic networks. arXiv:1503.05571.
5. Allen, R. B. (1987). Several studies on natural language and back-propagation. In IEEE First International Conference on Neural Networks, volume 2, pages 335–341, San Diego. http://boballen.info/RBA/PAPERS/NL-BP/nl-bp.pdf.
6. Anderson, E. (1935). The Irises of the Gaspé Peninsula. Bulletin of the American Iris Society, 59, 2–5.
7. Ba, J., Mnih, V., and Kavukcuoglu, K. (2014). Multiple object recognition with visual attention. arXiv:1412.7755 .
8. Bachman, P. and Precup, D. (2015). Variational generative stochastic networks with collaborative shaping. In Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6–11 July 2015, pages 1964–1972.
9. Bacon, P.-L., Bengio, E., Pineau, J., and Precup, D. (2015). Conditional computation in neural networks using a decision-theoretic approach. In 2nd Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM 2015).
10. Bagnell, J. A. and Bradley, D. M. (2009). Differentiable sparse coding. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, Advances in Neural Information Processing Systems 21 (NIPS'08), pages 113–120.
11. Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In ICLR'2015, arXiv:1409.0473.
12. Bahl, L. R., Brown, P., de Souza, P. V., and Mercer, R. L. (1987). Speech recognition with continuous-parameter hidden Markov models. Computer, Speech and Language, 2, 219–234.
13. Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. Neural Networks, 2, 53–58.
14. Baldi, P., Brunak, S., Frasconi, P., Soda, G., and Pollastri, G. (1999). Exploiting the past and the future in protein secondary structure prediction. Bioinformatics, 15(11), 937–946.
15. Baldi, P., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. Nature communications, 5.
16. Ballard, D. H., Hinton, G. E., and Sejnowski, T. J. (1983). Parallel vision computation. Nature.

17. Barlow, H. B. (1989). Unsupervised learning. Neural Computation, 1, 295–311.
18. Barron, A. E. (1993). Universal approximation bounds for superpositions of a sigmoidal function. IEEE Trans. on Information Theory, 39, 930–945.
19. Bartholomew, D. J. (1987). Latent variable models and factor analysis. Oxford University Press.
20. Basilevsky, A. (1994). Statistical Factor Analysis and Related Methods: Theory and Applications. Wiley.
21. Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.
22. Basu, S. and Christensen, J. (2013). Teaching classification boundaries to humans. In AAAI'2013.
23. Baxter, J. (1995). Learning internal representations. In Proceedings of the 8th International Conference on Computational Learning Theory (COLT'95), pages 311–320, Santa Cruz, California. ACM Press.
24. Bayer, J. and Osendorfer, C. (2014). Learning stochastic recurrent networks. ArXiv e-prints.
25. Becker, S. and Hinton, G. (1992). A self-organizing neural network that discovers surfaces in random-dot stereograms. Nature, 355, 161–163.
26. Behnke, S. (2001). Learning iterative image reconstruction in the neural abstraction pyramid. Int. J. Computational Intelligence and Applications, 1(4), 427–438.
27. Beiu, V., Quintana, J. M., and Avedillo, M. J. (2003). VLSI implementations of threshold logic-a comprehensive survey. Neural Networks, IEEE Transactions on, 14(5), 1217–1243.
28. Belkin, M. and Niyogi, P. (2002). Laplacian eigenmaps and spectral techniques for embedding and clustering. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, Advances in Neural Information Processing Systems 14 (NIPS'01), Cambridge, MA. MIT Press.
29. Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. Neural Computation, 15(6), 1373–1396.
30. Bengio, E., Bacon, P.-L., Pineau, J., and Precup, D. (2015a). Conditional computation in neural networks for faster models. arXiv:1511.06297.
31. Bengio, S. and Bengio, Y. (2000a). Taking on the curse of dimensionality in joint distributions using neural networks. IEEE Transactions on Neural Networks, special issue on Data Mining and Knowledge Discovery, 11(3), 550–557.
32. Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015b). Scheduled sampling for sequence prediction with recurrent neural networks. Technical report, arXiv:1506.03099.
33. Bengio, Y. (1991). Artificial Neural Networks and their Application to Sequence Recognition. Ph.D. thesis, McGill University, (Computer Science), Montreal, Canada.
34. Bengio, Y. (2000). Gradient-based optimization of hyperparameters. Neural Computation, 12(8), 1889–1900.
35. Bengio, Y. (2002). New distributed probabilistic language models. Technical Report 1215, Dept. IRO, Université de Montréal.
36. Bengio, Y. (2009). Learning deep architectures for AI. Now Publishers.
37. Bengio, Y. (2013). Deep learning of representations: looking forward. In Statistical Language and Speech Processing, volume 7978 of Lecture Notes in Computer Science, pages 1–37. Springer, also in arXiv at http://arxiv.org/abs/1305.0445.

38. Bengio, Y. (2015). Early inference in energy-based models approximates back-propagation. Technical Report arXiv:1510.02777, Universite de Montreal.
39. Bengio, Y. and Bengio, S. (2000b). Modeling high-dimensional discrete data with multilayer neural networks. In NIPS 12, pages 400–406. MIT Press.
40. Bengio, Y. and Delalleau, O. (2009). Justifying and generalizing contrastive divergence. Neural Computation, 21(6), 1601–1621.
41. Bengio, Y. and Grandvalet, Y. (2004). No unbiased estimator of the variance of k-fold cross-validation. In S. Thrun, L. Saul, and B. Schölkopf, editors, Advances in Neural Information Processing Systems 16 (NIPS'03), Cambridge, MA. MIT Press, Cambridge.
42. Bengio, Y. and LeCun, Y. (2007). Scaling learning algorithms towards AI. In Large Scale Kernel Machines.
43. Bengio, Y. and Monperrus, M. (2005). Non-local manifold tangent learning. In L. Saul, Y. Weiss, and L. Bottou, editors, Advances in Neural Information Processing Systems 17 (NIPS'04), pages 129–136. MIT Press.
44. Bengio, Y. and Sénécal, J.-S. (2003). Quick training of probabilistic neural nets by importance sampling. In Proceedings of AISTATS 2003.
45. Bengio, Y. and Sénécal, J.-S. (2008). Adaptive importance sampling to accelerate training of a neural probabilistic language model. IEEE Trans. Neural Networks, 19(4), 713–722.
46. Bengio, Y., De Mori, R., Flammia, G., and Kompe, R. (1991). Phonetically motivated acoustic parameters for continuous speech recognition using artificial neural networks. In Proceedings of EuroSpeech'91.
47. Bengio, Y., De Mori, R., Flammia, G., and Kompe, R. (1992). Neural network-Gaussian mixture hybrid for speech recognition or density estimation. In NIPS 4, pages 175–182. Morgan Kaufmann.
48. Bengio, Y., Frasconi, P., and Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In IEEE International Conference on Neural Networks, pages 1183–1195, San Francisco. IEEE Press. (invited paper).
49. Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Tr. Neural Nets.
50. Bengio, Y., Latendresse, S., and Dugas, C. (1999). Gradient-based learning of hyper-parameters. Learning Conference, Snowbird.
51. Bengio, Y., Ducharme, R., and Vincent, P. (2001). A neural probabilistic language model. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, NIPS'2000, pages 932–938. MIT Press.
52. Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. JMLR, 3, 1137–1155.
53. Bengio, Y., Le Roux, N., Vincent, P., Delalleau, O., and Marcotte, P. (2006a). Convex neural networks. In NIPS'2005, pages 123–130.
54. Bengio, Y., Delalleau, O., and Le Roux, N. (2006b). The curse of highly variable functions for local kernel machines. In NIPS'2005.
55. Bengio, Y., Larochelle, H., and Vincent, P. (2006c). Non-local manifold Parzen windows. In NIPS'2005. MIT Press.
56. Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In NIPS'2006.
57. Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In ICML'09.

58. Bengio, Y., Mesnil, G., Dauphin, Y., and Rifai, S. (2013a). Better mixing via deep representations. In ICML'2013.
59. Bengio, Y., Léonard, N., and Courville, A. (2013b). Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv:1308.3432.
60. Bengio, Y., Yao, L., Alain, G., and Vincent, P. (2013c). Generalized denoising auto-encoders as generative models. In NIPS'2013.
61. Bengio, Y., Courville, A., and Vincent, P. (2013d). Representation learning: A review and new perspectives. IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI), 35(8), 1798–1828.
62. Bengio, Y., Thibodeau-Laufer, E., Alain, G., and Yosinski, J. (2014). Deep generative stochastic networks trainable by backprop. In ICML'2014.
63. Bennett, C. (1976). Efficient estimation of free energy differences from Monte Carlo data. Journal of Computational Physics, 22(2), 245–268.
64. Bennett, J. and Lanning, S. (2007). The Netflix prize.
65. Berger, A. L., Della Pietra, V. J., and Della Pietra, S. A. (1996). A maximum entropy approach to natural language processing. Computational Linguistics, 22, 39–71.
66. Berglund, M. and Raiko, T. (2013). Stochastic gradient estimate variance in contrastive divergence and persistent contrastive divergence. CoRR, abs/1312.6002.
67. Bergstra, J. (2011). Incorporating Complex Cells into Neural Networks for Pattern Classification. Ph.D. thesis, Université de Montréal.
68. Bergstra, J. and Bengio, Y. (2009). Slow, decorrelated features for pretraining complex cell-like networks. In NIPS'2009.
69. Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. J. Machine Learning Res., 13, 281–305.
70. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In Proc. SciPy.
71. Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In NIPS'2011.
72. Berkes, P. and Wiskott, L. (2005). Slow feature analysis yields a rich repertoire of complex cell properties. Journal of Vision, 5(6), 579–602.
73. Bertsekas, D. P. and Tsitsiklis, J. (1996). Neuro-Dynamic Programming. Athena Scientific.
74. Besag, J. (1975). Statistical analysis of non-lattice data. The Statistician, 24(3), 179–195.
75. Bishop, C. M. (1994). Mixture density networks.
76. Bishop, C. M. (1995a). Regularization and complexity control in feed-forward networks. In Proceedings International Conference on Artificial Neural Networks ICANN'95, volume 1, page 141–148.
77. Bishop, C. M. (1995b). Training with noise is equivalent to Tikhonov regularization. Neural Computation, 7(1), 108–116.
78. Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.
79. Blum, A. L. and Rivest, R. L. (1992). Training a 3-node neural network is NP-complete.
80. Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1989). Learnability and the Vapnik–Chervonenkis dimension. Journal of the ACM, 36(4), 929–865.
81. Bonnet, G. (1964). Transformations des signaux aléatoires à travers les systèmes non linéaires sans mémoire. Annales des Télécommunications, 19(9–10), 203–220.

82. Bordes, A., Weston, J., Collobert, R., and Bengio, Y. (2011). Learning structured embeddings of knowledge bases. In AAAI 2011.
83. Bordes, A., Glorot, X., Weston, J., and Bengio, Y. (2012). Joint learning of words and meaning representations for open-text semantic parsing. AISTATS'2012.
84. Bordes, A., Glorot, X., Weston, J., and Bengio, Y. (2013a). A semantic matching energy function for learning with multi-relational data. Machine Learning: Special Issue on Learning Semantics.
85. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. (2013b). Translating embeddings for modeling multi-relational data. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 2787–2795. Curran Associates, Inc.
86. Bornschein, J. and Bengio, Y. (2015). Reweighted wake-sleep. In ICLR'2015, arXiv:1406.2751.
87. Bornschein, J., Shabanian, S., Fischer, A., and Bengio, Y. (2015). Training bidirectional Helmholtz machines. Technical report, arXiv:1506.03877.
88. Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In COLT '92: Proceedings of the fifth annual workshop on Computational learning theory, pages 144–152, New York, NY, USA. ACM.
89. Bottou, L. (1998). Online algorithms and stochastic approximations. In D. Saad, editor, Online Learning in Neural Networks. Cambridge University Press, Cambridge, UK.
90. Bottou, L. (2011). From machine learning to machine reasoning. Technical report, arXiv.1102.1808.
91. Bottou, L. (2015). Multilayer neural networks. Deep Learning Summer School.
92. Bottou, L. and Bousquet, O. (2008). The tradeoffs of large scale learning. In NIPS'2008.
93. Boulanger-Lewandowski, N., Bengio, Y., and Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In ICML'12.
94. Boureau, Y., Ponce, J., and LeCun, Y. (2010). A theoretical analysis of feature pooling in vision algorithms. In Proc. International Conference on Machine learning (ICML'10).
95. Boureau, Y., Le Roux, N., Bach, F., Ponce, J., and LeCun, Y. (2011). Ask the locals: multi-way local pooling for image recognition. In Proc. International Conference on Computer Vision (ICCV'11). IEEE.
96. Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. Biological Cybernetics, 59, 291–294.
97. Bourlard, H. and Wellekens, C. (1989). Speech pattern discrimination and multilayered perceptrons. Computer Speech and Language, 3, 1–19.
98. Boyd, S. and Vandenberghe, L. (2004). Convex Optimization. Cambridge University Press, New York, NY, USA.
99. Brady, M. L., Raghavan, R., and Slawny, J. (1989). Back-propagation fails to separate where perceptrons succeed. IEEE Transactions on Circuits and Systems, 36, 665–674.
100. Brakel, P., Stroobandt, D., and Schrauwen, B. (2013). Training energy-based models for time-series imputation. Journal of Machine Learning Research, 14, 2771–2797.
101. Brand, M. (2003). Charting a manifold. In NIPS'2002, pages 961–968. MIT Press.
102. Breiman, L. (1994). Bagging predictors. Machine Learning, 24(2), 123–140.
103. Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). Classification and Regression Trees. Wadsworth International Group, Belmont, CA.

104. Bridle, J. S. (1990). Alphanets: a recurrent ‘neural’ network architecture with a hidden Markov model interpretation. Speech Communication, 9(1), 83–92.
105. Briggman, K., Denk, W., Seung, S., Helmstaedter, M. N., and Turaga, S. C. (2009). Maximin affinity learning of image segmentation. In NIPS’2009, pages 1865–1873.
106. Brown, P. F., Cocke, J., Pietra, S. A. D., Pietra, V. J. D., Jelinek, F., Lafferty, J. D., Mercer, R. L., and Roossin, P. S. (1990). A statistical approach to machine translation. Computational linguistics, 16(2), 79–85.
107. Brown, P. F., Pietra, V. J. D., DeSouza, P. V., Lai, J.C., and Mercer, R. L. (1992). Class-based n-gram models of natural language. Computational Linguistics, 18, 467–479.
108. Bryson, A. and Ho, Y. (1969). Applied optimal control: optimization, estimation, and control. Blaisdell Pub. Co.
109. Bryson, Jr., A. E. and Denham, W. F. (1961). A steepest-ascent method for solving optimum programming problems. Technical Report BR-1303, Raytheon Company, Missle and Space Division.
110. Buciluǎ, C., Caruana, R., and Niculescu-Mizil, A. (2006). Model compression. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 535–541. ACM.
111. Burda, Y., Grosse, R., and Salakhutdinov, R. (2015). Importance weighted auto-encoders. arXiv preprint arXiv:1509.00519.
112. Cai, M., Shi, Y., and Liu, J. (2013). Deep maxout neural networks for speech recognition. In Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on, pages 291–296. IEEE.
113. Carreira-Perpiñan, M. A. and Hinton, G. E. (2005). On contrastive divergence learning. In R. G. Cowell and Z. Ghahramani, editors, Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS’05), pages 33–40. Society for Artificial Intelligence and Statistics.
114. Caruana, R. (1993). Multitask connectionist learning. In Proc. 1993 Connectionist Models Summer School, pages 372–379.
115. Cauchy, A. (1847). Méthode générale pour la résolution de systèmes d’équations simultanées. In Compte rendu des séances de l’académie des sciences, pages 536–538.
116. Cayton, L. (2005). Algorithms for manifold learning. Technical Report CS2008-0923, UCSD.
117. Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. ACM computing surveys (CSUR), 41(3), 15.
118. Chapelle, O., Weston, J., and Schölkopf, B. (2003). Cluster kernels for semi-supervised learning. In S. Becker, S. Thrun, and K. Obermayer, editors, Advances in Neural Information Processing Systems 15 (NIPS’02), pages 585–592, Cambridge, MA. MIT Press.
119. Chapelle, O., Schölkopf, B., and Zien, A., editors (2006). Semi-Supervised Learning. MIT Press, Cambridge, MA.
120. Chellapilla, K., Puri, S., and Simard, P. (2006). High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule (France). Université de Rennes 1, Suvisoft. http://www.suvisoft.com.
121. Chen, B., Ting, J.-A., Marlin, B. M., and de Freitas, N. (2010). Deep learning of invariant spatio-temporal features from video. NIPS*2010 Deep Learning and Unsupervised Feature Learning Workshop.

122. Chen, S. F. and Goodman, J. T. (1999). An empirical study of smoothing techniques for language modeling. Computer, Speech and Language, 13(4), 359–393.
123. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014a). DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, pages 269–284. ACM.
124. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274.
125. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. (2014b). DaDianNao: A machine-learning supercomputer. In Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on, pages 609–622. IEEE.
126. Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. (2014). Project Adam: Building an efficient and scalable deep learning training system. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14).
127. Cho, K., Raiko, T., and Ilin, A. (2010). Parallel tempering is efficient for learning restricted Boltzmann machines. In IJCNN'2010.
128. Cho, K., Raiko, T., and Ilin, A. (2011). Enhanced gradient and adaptive learning rate for training restricted Boltzmann machines. In ICML'2011, pages 105–112.
129. Cho, K., van Merriënboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014a). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014).
130. Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014b). On the properties of neural machine translation: Encoder-decoder approaches. ArXiv e-prints, abs/1409.1259.
131. Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surface of multilayer networks.
132. Chorowski, J., Bahdanau, D., Cho, K., and Bengio, Y. (2014). End-to-end continuous speech recognition using attention-based recurrent NN: First results. arXiv:1412.1602.
133. Chrisman, L. (1991). Learning recursive distributed representations for holistic computation. Connection Science, 3(4), 345–366. http://repository.cmu.edu/cgi/viewcontent.cgi?article=3061&context=compsci.
134. Christianson, B. (1992). Automatic Hessians by reverse accumulation. IMA Journal of Numerical Analysis, 12(2), 135–150.
135. Chrupala, G., Kadar, A., and Alishahi, A. (2015). Learning language through pictures. arXiv 1506.03694.
136. Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. NIPS'2014 Deep Learning workshop, arXiv 1412.3555.
137. Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2015a). Gated feedback recurrent neural networks. In ICML'15.
138. Chung, J., Kastner, K., Dinh, L., Goel, K., Courville, A., and Bengio, Y. (2015b). A recurrent latent variable model for sequential data. In NIPS'2015.

139. Ciresan, D., Meier, U., Masci, J., and Schmidhuber, J. (2012). Multi-column deep neural network for traffic sign classification. Neural Networks, 32, 333–338.
140. Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep big simple neural nets for handwritten digit recognition. Neural Computation, 22, 1–14.
141. Coates, A. and Ng, A. Y. (2011). The importance of encoding versus training with sparse coding and vector quantization. In ICML'2011.
142. Coates, A., Lee, H., and Ng, A. Y. (2011). An analysis of single-layer networks in unsupervised feature learning. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011).
143. Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with COTS HPC systems. In S. Dasgupta and D. McAllester, editors, Proceedings of the 30th International Conference on Machine Learning (ICML-13), volume 28 (3), pages 1337–1345. JMLR Workshop and Conference Proceedings.
144. Cohen, N., Sharir, O., and Shashua, A. (2015). On the expressive power of deep learning: A tensor analysis. arXiv:1509.05009.
145. Collobert, R. (2004). Large Scale Machine Learning. Ph.D. thesis, Université de Paris VI, LIP6.
146. Collobert, R. (2011). Deep learning for efficient discriminative parsing. In AISTATS'2011.
147. Collobert, R. and Weston, J. (2008a). A unified architecture for natural language processing: Deep neural networks with multitask learning. In ICML'2008.
148. Collobert, R. and Weston, J. (2008b). A unified architecture for natural language processing: Deep neural networks with multitask learning. In ICML'2008.
149. Collobert, R., Bengio, S., and Bengio, Y. (2001). A parallel mixture of SVMs for very large scale problems. Technical Report IDIAP-RR-01-12, IDIAP.
150. Collobert, R., Bengio, S., and Bengio, Y. (2002). Parallel mixture of SVMs for very large scale problems. Neural Computation, 14(5), 1105–1114.
151. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011a). Natural language processing (almost) from scratch. The Journal of Machine Learning Research, 12, 2493–2537.
152. Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011b). Torch7: A Matlab-like environment for machine learning. In BigLearn, NIPS Workshop.
153. Comon, P. (1994). Independent component analysis – a new concept? Signal Processing, 36, 287–314.
154. Cortes, C. and Vapnik, V. (1995). Support vector networks. Machine Learning, 20, 273–297.
155. Couprie, C., Farabet, C., Najman, L., and LeCun, Y. (2013). Indoor semantic segmentation using depth information. In International Conference on Learning Representations (ICLR2013).
156. Courbariaux, M., Bengio, Y., and David, J.-P. (2015). Low precision arithmetic for deep learning. In Arxiv:1412.7024, ICLR'2015 Workshop.
157. Courville, A., Bergstra, J., and Bengio, Y. (2011). Unsupervised models of images by spike-and-slab RBMs. In ICML'11.
158. Courville, A., Desjardins, G., Bergstra, J., and Bengio, Y. (2014). The spike-and-slab RBM and extensions to discrete and sparse data distributions. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 36(9), 1874–1887.

159. Cover, T. M. and Thomas, J. A. (2006). Elements of Information Theory, 2nd Edition. Wiley-Interscience.
160. Cox, D. and Pinto, N. (2011). Beyond simple features: A large-scale feature search approach to unconstrained face recognition. In Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on, pages 8–15. IEEE.
161. Cramér, H. (1946). Mathematical methods of statistics. Princeton University Press.
162. Crick, F. H. C. and Mitchison, G. (1983). The function of dream sleep. Nature, 304, 111–114.
163. Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems, 2, 303–314.
164. Dahl, G. E., Ranzato, M., Mohamed, A., and Hinton, G. E. (2010). Phone recognition with the mean-covariance restricted Boltzmann machine. In NIPS'2010.
165. Dahl, G. E., Yu, D., Deng, L., and Acero, A. (2012). Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. IEEE Transactions on Audio, Speech, and Language Processing, 20(1), 33–42.
166. Dahl, G. E., Sainath, T. N., and Hinton, G. E. (2013). Improving deep neural networks for LVCSR using rectified linear units and dropout. In ICASSP'2013.
167. Dahl, G. E., Jaitly, N., and Salakhutdinov, R. (2014). Multi-task neural networks for QSAR predictions. arXiv:1406.1231.
168. Dauphin, Y. and Bengio, Y. (2013). Stochastic ratio matching of RBMs for sparse high-dimensional inputs. In NIPS26. NIPS Foundation.
169. Dauphin, Y., Glorot, X., and Bengio, Y. (2011). Large-scale learning of embeddings with reconstruction sampling. In ICML'2011.
170. Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In NIPS'2014.
171. Davis, A., Rubinstein, M., Wadhwa, N., Mysore, G., Durand, F., and Freeman, W. T. (2014). The visual microphone: Passive recovery of sound from video. ACM Transactions on Graphics (Proc. SIGGRAPH), 33(4), 79:1–79:10.
172. Dayan, P. (1990). Reinforcement comparison. In Connectionist Models: Proceedings of the 1990 Connectionist Summer School, San Mateo, CA.
173. Dayan, P. and Hinton, G. E. (1996). Varieties of Helmholtz machine. Neural Networks, 9(8), 1385–1403.
174. Dayan, P., Hinton, G. E., Neal, R. M., and Zemel, R. S. (1995). The Helmholtz machine. Neural computation, 7(5), 889–904.
175. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large scale distributed deep networks. In NIPS'2012.
176. Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. Computational Intelligence, 5(3), 142–150.
177. Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), 391–407.
178. Delalleau, O. and Bengio, Y. (2011). Shallow vs. deep sum-product networks. In NIPS.
179. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09.

180. Deng, J., Berg, A. C., Li, K., and Fei-Fei, L. (2010a). What does classifying more than 10,000 image categories tell us? In Proceedings of the 11th European Conference on Computer Vision: Part V, ECCV'10, pages 71–84, Berlin, Heidelberg. Springer-Verlag.
181. Deng, L. and Yu, D. (2014). Deep learning – methods and applications. Foundations and Trends in Signal Processing.
182. Deng, L., Seltzer, M., Yu, D., Acero, A., Mohamed, A., and Hinton, G. (2010b). Binary coding of speech spectrograms using a deep auto-encoder. In Interspeech 2010, Makuhari, Chiba, Japan.
183. Denil, M., Bazzani, L., Larochelle, H., and de Freitas, N. (2012). Learning where to attend with deep architectures for image tracking. Neural Computation, 24(8), 2151–2184.
184. Denton, E., Chintala, S., Szlam, A., and Fergus, R. (2015). Deep generative image models using a Laplacian pyramid of adversarial networks. NIPS.
185. Desjardins, G. and Bengio, Y. (2008). Empirical evaluation of convolutional RBMs for vision. Technical Report 1327, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal.
186. Desjardins, G., Courville, A. C., Bengio, Y., Vincent, P., and Delalleau, O. (2010). Tempered Markov chain Monte Carlo for training of restricted Boltzmann machines. In International Conference on Artificial Intelligence and Statistics, pages 145–152.
187. Desjardins, G., Courville, A., and Bengio, Y. (2011). On tracking the partition function. In NIPS'2011.
188. Desjardins, G., Simonyan, K., Pascanu, R., et al. (2015). Natural neural networks. In Advances in Neural Information Processing Systems, pages 2062–2070.
189. Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R., and Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In Proc. ACL'2014.
190. Devroye, L. (2013). Non-Uniform Random Variate Generation. SpringerLink: Bücher. Springer New York.
191. DiCarlo, J. J. (2013). Mechanisms underlying visual object recognition: Humans vs. neurons vs. machines. NIPS Tutorial.
192. Dinh, L., Krueger, D., and Bengio, Y. (2014). NICE: Non-linear independent components estimation. arXiv:1410.8516.
193. Donahue, J., Hendricks, L. A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., and Darrell, T. (2014). Long-term recurrent convolutional networks for visual recognition and description. arXiv:1411.4389.
194. Donoho, D. L. and Grimes, C. (2003). Hessian eigenmaps: new locally linear embedding techniques for high-dimensional data. Technical Report 2003-08, Dept. Statistics, Stanford University.
195. Dosovitskiy, A., Springenberg, J. T., and Brox, T. (2015). Learning to generate chairs with convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1538–1546.
196. Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. IEEE Transactions on Neural Networks, 1, 75–80.
197. Dreyfus, S. E. (1962). The numerical solution of variational problems. Journal of Mathematical Analysis and Applications, 5(1), 30–45.

198. Dreyfus, S. E. (1973). The computational solution of optimal control problems with time lag. IEEE Transactions on Automatic Control, 18(4), 383–385.
199. Drucker, H. and LeCun, Y. (1992). Improving generalisation performance using double back-propagation. IEEE Transactions on Neural Networks, 3(6), 991–997.
200. Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research.
201. Dudik, M., Langford, J., and Li, L. (2011). Doubly robust policy evaluation and learning. In Proceedings of the 28th International Conference on Machine learning, ICML '11.
202. Dugas, C., Bengio, Y., Bélisle, F., and Nadeau, C. (2001). Incorporating second-order functional knowledge for better option pricing. In T. Leen, T. Dietterich, and V. Tresp, editors, Advances in Neural Information Processing Systems 13 (NIPS'00), pages 472–478. MIT Press.
203. Dziugaite, G. K., Roy, D. M., and Ghahramani, Z. (2015). Training generative neural networks via maximum mean discrepancy optimization. arXiv preprint arXiv:1505.03906.
204. El Hihi, S. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In NIPS'1995.
205. Elkahky, A. M., Song, Y., and He, X. (2015). A multi-view deep learning approach for cross domain user modeling in recommendation systems. In Proceedings of the 24th International Conference on World Wide Web, pages 278–288.
206. Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. Cognition, 48, 781–799.
207. Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In Proceedings of AISTATS'2009.
208. Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? J. Machine Learning Res.
209. Fahlman, S. E., Hinton, G. E., and Sejnowski, T. J. (1983). Massively parallel architectures for AI: NETL, thistle, and Boltzmann machines. In Proceedings of the National Conference on Artificial Intelligence AAAI-83.
210. Fang, H., Gupta, S., Iandola, F., Srivastava, R., Deng, L., Dollár, P., Gao, J., He, X., Mitchell, M., Platt, J. C., Zitnick, C. L., and Zweig, G. (2015). From captions to visual concepts and back. arXiv:1411.4952.
211. Farabet, C., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., and Talay, S. (2011). Large-scale FPGA-based convolutional networks. In R. Bekkerman, M. Bilenko, and J. Langford, editors, Scaling up Machine Learning: Parallel and Distributed Approaches. Cambridge University Press.
212. Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2013). Learning hierarchical features for scene labeling. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8), 1915–1929.
213. Fei-Fei, L., Fergus, R., and Perona, P. (2006). One-shot learning of object categories. IEEE Transactions on Pattern Analysis and Machine Intelligence, 28(4), 594–611.
214. Finn, C., Tan, X. Y., Duan, Y., Darrell, T., Levine, S., and Abbeel, P. (2015). Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. arXiv preprint arXiv:1509.06113.
215. Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. Annals of Eugenics, 7, 179–188.

216. Földiák, P. (1989). Adaptive network for optimal linear feature extraction. In International Joint Conference on Neural Networks (IJCNN), volume 1, pages 401–405, Washington 1989. IEEE, New York.
217. Forcada, M., and Teco, R. (1997). Recursive hetero-associative memories for translation. In Biological and Artificial Computation: From Neuroscience to Technology, pages 453–462. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.1968.
218. Franzius, M., Sprekeler, H., and Wiskott, L. (2007). Slowness and sparseness lead to place, head-direction, and spatial-view cells.
219. Franzius, M., Wilbert, N., and Wiskott, L. (2008). Invariant object recognition with slow feature analysis. In Artificial Neural Networks-ICANN 2008, pages 961–970. Springer.
220. Frasconi, P., Gori, M., and Sperduti, A. (1997). On the efficient classification of data structures by neural networks. In Proc. Int. Joint Conf. on Artificial Intelligence.
221. Frasconi, P., Gori, M., and Sperduti, A. (1998). A general framework for adaptive processing of data structures. IEEE Transactions on Neural Networks, 9(5), 768–786.
222. Freund, Y. and Schapire, R. E. (1996a). Experiments with a new boosting algorithm. In Machine Learning: Proceedings of Thirteenth International Conference, pages 148–156, USA. ACM.
223. Freund, Y. and Schapire, R. E. (1996b). Game theory, on-line prediction and boosting. In Proceedings of the Ninth Annual Conference on Computational Learning Theory, pages 325–332.
224. Frey, B. J. (1998). Graphical models for machine learning and digital communication. MIT Press.
225. Frey, B. J., Hinton, G. E., and Dayan, P. (1996). Does the wake-sleep algorithm learn good density estimators? In D. Touretzky, M. Mozer, and M. Hasselmo, editors, Advances in Neural Information Processing Systems 8 (NIPS'95), pages 661–670. MIT Press, Cambridge, MA.
226. Frobenius, G. (1908). Über matrizen aus positiven elementen, s. B. Preuss. Akad. Wiss. Berlin, Germany.
227. Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. Biological Cybernetics, 20, 121–136.
228. Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36, 193–202.
229. Gal, Y. and Ghahramani, Z. (2015). Bayesian convolutional neural networks with Bernoulli approximate variational inference. arXiv preprint arXiv:1506.02158.
230. Gallinari, P., LeCun, Y., Thiria, S., and Fogelman-Soulie, F. (1987). Memoires associatives distribuees. In Proceedings of COGNITIVA 87, Paris, La Villette.
231. Garcia-Duran, A., Bordes, A., Usunier, N., and Grandvalet, Y. (2015). Combining two and three-way embeddings models for link prediction in knowledge bases. arXiv preprint arXiv:1506.00999.
232. Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., and Pallett, D. S. (1993). Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. NASA STI/Recon Technical Report N, 93, 27403.
233. Garson, J. (1900). The metric system of identification of criminals, as used in Great Britain and Ireland. The Journal of the Anthropological Institute of Great Britain and Ireland, (2), 177–227.

234. Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. Neural computation, 12(10), 2451–2471.
235. Ghahramani, Z. and Hinton, G. E. (1996). The EM algorithm for mixtures of factor analyzers. Technical Report CRG-TR-96-1, Dpt. of Comp. Sci., Univ. of Toronto.
236. Gillick, D., Brunk, C., Vinyals, O., and Subramanya, A. (2015). Multilingual language processing from bytes. arXiv preprint arXiv:1512.00103.
237. Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2015). Region-based convolutional networks for accurate object detection and segmentation.
238. Giudice, M. D., Manera, V., and Keysers, C. (2009). Programmed to learn? The ontogeny of mirror neurons. Dev. Sci., 12(2), 350–363.
239. Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In AISTATS'2010.
240. Glorot, X., Bordes, A., and Bengio, Y. (2011a). Deep sparse rectifier neural networks. In AISTATS'2011.
241. Glorot, X., Bordes, A., and Bengio, Y. (2011b). Domain adaptation for large-scale sentiment classification: A deep learning approach. In ICML'2011.
242. Goldberger, J., Roweis, S., Hinton, G. E., and Salakhutdinov, R. (2005). Neighbourhood components analysis. In L. Saul, Y. Weiss, and L. Bottou, editors, Advances in Neural Information Processing Systems 17 (NIPS'04). MIT Press.
243. Gong, S., McKenna, S., and Psarrou, A. (2000). Dynamic Vision: From Images to Face Recognition. Imperial College Press.
244. Goodfellow, I., Le, Q., Saxe, A., and Ng, A. (2009). Measuring invariances in deep networks. In NIPS'2009, pages 646–654.
245. Goodfellow, I., Koenig, N., Muja, M., Pantofaru, C., Sorokin, A., and Takayama, L. (2010). Help me help you: Interfaces for personal robots. In Proc. of Human Robot Interaction (HRI), Osaka, Japan. ACM Press, ACM Press.
246. Goodfellow, I. J. (2010). Technical report: Multidimensional, downsampled convolution for autoencoders. Technical report, Université de Montréal.
247. Goodfellow, I. J. (2014). On distinguishability criteria for estimating generative models. In International Conference on Learning Representations, Workshops Track.
248. Goodfellow, I. J., Courville, A., and Bengio, Y. (2011). Spike-and-slab sparse coding for unsupervised feature discovery. In NIPS Workshop on Challenges in Learning Hierarchical Models.
249. Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013a). Maxout networks. In S. Dasgupta and D. McAllester, editors, ICML'13, pages 1319–1327.
250. Goodfellow, I. J., Mirza, M., Courville, A., and Bengio, Y. (2013b). Multi-prediction deep Boltzmann machines. In NIPS26. NIPS Foundation.
251. Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013c). Pylearn2: a machine learning research library. arXiv preprint arXiv:1308.4214.
252. Goodfellow, I. J., Courville, A., and Bengio, Y. (2013d). Scaling up spike-and-slab models for unsupervised feature learning. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8), 1902–1914.
253. Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. (2014a). An empirical investigation of catastrophic forgetting in gradient-based neural networks. In ICLR'2014.

254. Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014b). Explaining and harnessing adversarial examples. CoRR, abs/1412.6572.
255. Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014c). Generative adversarial networks. In NIPS'2014.
256. Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. (2014d). Multi-digit number recognition from Street View imagery using deep convolutional neural networks. In International Conference on Learning Representations.
257. Goodfellow, I. J., Vinyals, O., and Saxe, A. M. (2015). Qualitatively characterizing neural network optimization problems. In International Conference on Learning Representations.
258. Goodman, J. (2001). Classes for fast maximum entropy training. In International Conference on Acoustics, Speech and Signal Processing (ICASSP), Utah.
259. Gori, M. and Tesi, A. (1992). On the problem of local minima in backpropagation. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-14(1), 76–86.
260. Gosset, W. S. (1908). The probable error of a mean. Biometrika, 6(1), 1–25. Originally published under the pseudonym "Student".
261. Gouws, S., Bengio, Y., and Corrado, G. (2014). BilBOWA: Fast bilingual distributed representations without word alignments. Technical report, arXiv:1410.2455.
262. Graf, H. P. and Jackel, L. D. (1989). Analog electronic neural network circuits. Circuits and Devices Magazine, IEEE, 5(4), 44–49.
263. Graves, A. (2011). Practical variational inference for neural networks. In NIPS'2011.
264. Graves, A. (2012). Supervised Sequence Labelling with Recurrent Neural Networks. Studies in Computational Intelligence. Springer.
265. Graves, A. (2013). Generating sequences with recurrent neural networks. Technical report, arXiv:1308.0850.
266. Graves, A. and Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. In ICML'2014.
267. Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Networks, 18(5), 602–610.
268. Graves, A. and Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, NIPS'2008, pages 545–552.
269. Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In ICML'2006, pages 369–376, Pittsburgh, USA.
270. Graves, A., Liwicki, M., Bunke, H., Schmidhuber, J., and Fernández, S. (2008). Unconstrained on-line handwriting recognition with recurrent neural networks. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, NIPS'2007, pages 577–584.
271. Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. (2009). A novel connectionist system for unconstrained handwriting recognition. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 31(5), 855–868.
272. Graves, A., Mohamed, A., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In ICASSP'2013, pages 6645–6649.
273. Graves, A., Wayne, G., and Danihelka, I. (2014a). Neural Turing machines. arXiv:1410.5401.

274. Graves, A., Wayne, G., and Danihelka, I. (2014b). Neural Turing machines. arXiv preprint arXiv:1410.5401.
275. Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. (2015). Learning to transduce with unbounded memory. In NIPS'2015.
276. Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2015). LSTM: a search space odyssey. arXiv preprint arXiv:1503.04069.
277. Gregor, K. and LeCun, Y. (2010a). Emergence of complex-like cells in a temporal product network with local receptive fields. Technical report, arXiv:1006.0448.
278. Gregor, K. and LeCun, Y. (2010b). Learning fast approximations of sparse coding. In L. Bottou and M. Littman, editors, Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10). ACM.
279. Gregor, K., Danihelka, I., Mnih, A., Blundell, C., and Wierstra, D. (2014). Deep autoregressive networks. In International Conference on Machine Learning (ICML'2014).
280. Gregor, K., Danihelka, I., Graves, A., and Wierstra, D. (2015). DRAW: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623.
281. Gretton, A., Borgwardt, K. M., Rasch, M. J., Schölkopf, B., and Smola, A. (2012). A kernel two-sample test. The Journal of Machine Learning Research, 13(1), 723–773.
282. Gülçehre, Ç. and Bengio, Y. (2013). Knowledge matters: Importance of prior information for optimization. In International Conference on Learning Representations (ICLR'2013).
283. Guo, H. and Gelfand, S. B. (1992). Classification trees with neural network feature extraction. Neural Networks, IEEE Transactions on, 3(6), 923–933.
284. Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. CoRR, abs/1502.02551.
285. Gutmann, M. and Hyvarinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10).
286. Hadsell, R., Sermanet, P., Ben, J., Erkan, A., Han, J., Muller, U., and LeCun, Y. (2007). Online learning for offroad robots: Spatial label propagation to learn long-range traversability. In Proceedings of Robotics: Science and Systems, Atlanta, GA, USA.
287. Hajnal, A., Maass, W., Pudlak, P., Szegedy, M., and Turan, G. (1993). Threshold circuits of bounded depth. J. Comput. System. Sci., 46, 129–154.
288. Håstad, J. (1986). Almost optimal lower bounds for small depth circuits. In Proceedings of the 18th annual ACM Symposium on Theory of Computing, pages 6–20, Berkeley, California. ACM Press.
289. Håstad, J. and Goldmann, M. (1991). On the power of small-depth threshold circuits. Computational Complexity, 1, 113–129.
290. Hastie, T., Tibshirani, R., and Friedman, J. (2001). The elements of statistical learning: data mining, inference and prediction. Springer Series in Statistics. Springer Verlag.
291. He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. arXiv preprint arXiv:1502.01852.
292. Hebb, D. O. (1949). The Organization of Behavior. Wiley, New York.
293. Henaff, M., Jarrett, K., Kavukcuoglu, K., and LeCun, Y. (2011). Unsupervised learning of sparse features for scalable audio classification. In ISMIR'11.

294. Henderson, J. (2003). Inducing history representations for broad coverage statistical parsing. In HLT-NAACL, pages 103–110.
295. Henderson, J. (2004). Discriminative training of a neural network statistical parser. In Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics, page 95.
296. Henniges, M., Puertas, G., Bornschein, J., Eggert, J., and Lücke, J. (2010). Binary sparse coding. In Latent Variable Analysis and Signal Separation, pages 450–457. Springer.
297. Herault, J. and Ans, B. (1984). Circuits neuronaux à synapses modifiables: Décodage de messages composites par apprentissage non supervisé. Comptes Rendus de l'Académie des Sciences, 299(III-13), 525–528.
298. Hinton, G. (2012). Neural networks for machine learning. Coursera, video lectures.
299. Hinton, G., Deng, L., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012a). Deep neural networks for acoustic modeling in speech recognition. IEEE Signal Processing Magazine, 29(6), 82–97.
300. Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531.
301. Hinton, G. E. (1989). Connectionist learning procedures. Artificial Intelligence, 40, 185–234.
302. Hinton, G. E. (1990). Mapping part-whole hierarchies into connectionist networks. Artificial Intelligence, 46(1), 47–75.
303. Hinton, G. E. (1999). Products of experts. In ICANN'1999.
304. Hinton, G. E. (2000). Training products of experts by minimizing contrastive divergence. Technical Report GCNU TR 2000-004, Gatsby Unit, University College London.
305. Hinton, G. E. (2006). To recognize shapes, first learn to generate images. Technical Report UTML TR 2006-003, University of Toronto.
306. Hinton, G. E. (2007a). How to do backpropagation in a brain. Invited talk at the NIPS'2007 Deep Learning Workshop.
307. Hinton, G. E. (2007b). Learning multiple layers of representation. Trends in cognitive sciences, 11(10), 428–434.
308. Hinton, G. E. (2010). A practical guide to training restricted Boltzmann machines. Technical Report UTML TR 2010-003, Department of Computer Science, University of Toronto.
309. Hinton, G. E. and Ghahramani, Z. (1997). Generative models for discovering sparse distributed representations. Philosophical Transactions of the Royal Society of London.
310. Hinton, G. E. and McClelland, J. L. (1988). Learning representations by recirculation. In NIPS'1987, pages 358–366.
311. Hinton, G. E. and Roweis, S. (2003). Stochastic neighbor embedding. In NIPS'2002.
312. Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. Science, 313(5786), 504–507.
313. Hinton, G. E. and Sejnowski, T. J. (1986). Learning and relearning in Boltzmann machines. In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing, volume 1, chapter 7, pages 282–317. MIT Press, Cambridge.
314. Hinton, G. E. and Sejnowski, T. J. (1999). Unsupervised learning: foundations of neural computation. MIT press.

315. Hinton, G. E. and Shallice, T. (1991). Lesioning an attractor network: investigations of acquired dyslexia. Psychological review, 98(1), 74.
316. Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length, and Helmholtz free energy. In NIPS'1993.
317. Hinton, G. E., Sejnowski, T. J., and Ackley, D. H. (1984). Boltzmann machines: Constraint satisfaction networks that learn. Technical Report TR-CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer Science.
318. Hinton, G. E., McClelland, J., and Rumelhart, D. (1986). Distributed representations. In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, volume 1, pages 77–109. MIT Press, Cambridge.
319. Hinton, G. E., Revow, M., and Dayan, P. (1995a). Recognizing handwritten digits using mixtures of linear models. In G. Tesauro, D. Touretzky, and T. Leen, editors, Advances in Neural Information Processing Systems 7 (NIPS'94), pages 1015–1022. MIT Press, Cambridge, MA.
320. Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. M. (1995b). The wake-sleep algorithm for unsupervised neural networks. Science, 268, 1558–1161.
321. Hinton, G. E., Dayan, P., and Revow, M. (1997). Modelling the manifolds of images of handwritten digits. IEEE Transactions on Neural Networks, 8, 65–74.
322. Hinton, G. E., Welling, M., Teh, Y. W., and Osindero, S. (2001). A new view of ICA. In Proceedings of 3rd International Conference on Independent Component Analysis and Blind Signal Separation (ICA'01), pages 746–751, San Diego, CA.
323. Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. Neural Computation, 18, 1527–1554.
324. Hinton, G. E., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. (2012b). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Process. Mag., 29(6), 82–97.
325. Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012c). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580.
326. Hinton, G. E., Vinyals, O., and Dean, J. (2014). Dark knowledge. Invited talk at the BayLearn Bay Area Machine Learning Symposium.
327. Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, T. U. München.
328. Hochreiter, S. and Schmidhuber, J. (1995). Simplifying neural nets by discovering flat minima. In Advances in Neural Information Processing Systems 7, pages 529–536. MIT Press.
329. Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780.
330. Hochreiter, S., Bengio, Y., and Frasconi, P. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In J. Kolen and S. Kremer, editors, Field Guide to Dynamical Recurrent Networks. IEEE Press.
331. Holi, J. L. and Hwang, J.-N. (1993). Finite precision error analysis of neural network hardware implementations. Computers, IEEE Transactions on, 42(3), 281–290.
332. Holt, J. L. and Baker, T. E. (1991). Back propagation simulations using limited precision calculations. In Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on, volume 2, pages 121–126. IEEE.

333. Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2, 359–366.
334. Hornik, K., Stinchcombe, M., and White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. Neural networks, 3(5), 551–560.
335. Hsu, F.-H. (2002). Behind Deep Blue: Building the Computer That Defeated the World Chess Champion. Princeton University Press, Princeton, NJ, USA.
336. Huang, F. and Ogata, Y. (2002). Generalized pseudo-likelihood estimates for Markov random fields on lattice. Annals of the Institute of Statistical Mathematics, 54(1), 1–18.
337. Huang, P.-S., He, X., Gao, J., Deng, L., Acero, A., and Heck, L. (2013). Learning deep structured semantic models for web search using clickthrough data. In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management, pages 2333–2338. ACM.
338. Hubel, D. and Wiesel, T. (1968). Receptive fields and functional architecture of monkey striate cortex. Journal of Physiology (London), 195, 215–243.
339. Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. Journal of Physiology, 148, 574–591.
340. Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. Journal of Physiology (London), 160, 106–154.
341. Huszar, F. (2015). How (not) to train your generative model: schedule sampling, likelihood, adversary? arXiv:1511.05101.
342. Hutter, F., Hoos, H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In LION-5. Extended version as UBC Tech report TR-2010-10.
343. Hyotyniemi, H. (1996). Turing machines are recurrent neural networks. In STeP'96, pages 13–24.
344. Hyvärinen, A. (1999). Survey on independent component analysis. Neural Computing Surveys, 2, 94–128.
345. Hyvärinen, A. (2005). Estimation of non-normalized statistical models using score matching. Journal of Machine Learning Research, 6, 695–709.
346. Hyvärinen, A. (2007a). Connections between score matching, contrastive divergence, and pseudolikelihood for continuous-valued variables. IEEE Transactions on Neural Networks, 18, 1529–1531.
347. Hyvärinen, A. (2007b). Some extensions of score matching. Computational Statistics and Data Analysis, 51, 2499–2512.
348. Hyvärinen, A. and Hoyer, P. O. (1999). Emergence of topography and complex cell properties from natural images using extensions of ica. In NIPS, pages 827–833.
349. Hyvärinen, A. and Pajunen, P. (1999). Nonlinear independent component analysis: Existence and uniqueness results. Neural Networks, 12(3), 429–439.
350. Hyvärinen, A., Karhunen, J., and Oja, E. (2001a). Independent Component Analysis. Wiley-Interscience.
351. Hyvärinen, A., Hoyer, P. O., and Inki, M. O. (2001b). Topographic independent component analysis. Neural Computation, 13(7), 1527–1558.
352. Hyvärinen, A., Hurri, J., and Hoyer, P. O. (2009). Natural Image Statistics: A probabilistic approach to early computational vision. Springer-Verlag.

353. Iba, Y. (2001). Extended ensemble Monte Carlo. International Journal of Modern Physics, C12, 623–656.
354. Inayoshi, H. and Kurita, T. (2005). Improved generalization by adding both autoassociation and hidden-layer noise to neural-network-based-classifiers. IEEE Workshop on Machine Learning for Signal Processing, pages 141–146.
355. Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.
356. Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. Neural networks, 1(4), 295–307.
357. Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. Neural Computation, 3, 79–87.
358. Jaeger, H. (2003). Adaptive nonlinear system identification with echo state networks. In Advances in Neural Information Processing Systems 15.
359. Jaeger, H. (2007a). Discovering multiscale dynamical features with hierarchical echo state networks. Technical report, Jacobs University.
360. Jaeger, H. (2007b). Echo state network. Scholarpedia, 2(9), 2330.
361. Jaeger, H. (2012). Long short-term memory in echo state networks: Details of a simulation study. Technical report, Technical report, Jacobs University Bremen.
362. Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. Science, 304(5667), 78–80.
363. Jaeger, H., Lukosevicius, M., Popovici, D., and Siewert, U. (2007). Optimization and applications of echo state networks with leaky- integrator neurons. Neural Networks, 20(3), 335–352.
364. Jain, V., Murray, J. F., Roth, F., Turaga, S., Zhigulin, V., Briggman, K. L., Helmstaedter, M. N., Denk, W., and Seung, H. S. (2007). Supervised learning of image restoration with convolutional networks. In Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on, pages 1–8. IEEE.
365. Jaitly, N. and Hinton, G. (2011). Learning a better representation of speech soundwaves using restricted Boltzmann machines. In Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, pages 5884–5887. IEEE.
366. Jaitly, N. and Hinton, G. E. (2013). Vocal tract length perturbation (VTLP) improves speech recognition. In ICML'2013.
367. Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In ICCV'09.
368. Jarzynski, C. (1997). Nonequilibrium equality for free energy differences. Phys. Rev. Lett., 78, 2690–2693.
369. Jaynes, E. T. (2003). Probability Theory: The Logic of Science. Cambridge University Press.
370. Jean, S., Cho, K., Memisevic, R., and Bengio, Y. (2014). On using very large target vocabulary for neural machine translation. arXiv:1412.2007.
371. Jelinek, F. and Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In E. S. Gelsema and L. N. Kanal, editors, Pattern Recognition in Practice. North-Holland, Amsterdam.
372. Jia, Y. (2013). Caffe: An open source convolutional architecture for fast feature embedding. http://caffe.berkeleyvision.org/.

373. Jia, Y., Huang, C., and Darrell, T. (2012). Beyond spatial pyramids: Receptive field learning for pooled image features. In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, pages 3370–3377. IEEE.
374. Jim, K.-C., Giles, C. L., and Horne, B. G. (1996). An analysis of noise in recurrent neural networks: convergence and generalization. IEEE Transactions on Neural Networks, 7(6), 1424–1438.
375. Jordan, M. I. (1998). Learning in Graphical Models. Kluwer, Dordrecht, Netherlands.
376. Joulin, A. and Mikolov, T. (2015). Inferring algorithmic patterns with stack-augmented recurrent nets. arXiv preprint arXiv:1503.01007.
377. Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical evaluation of recurrent network architectures. In ICML'2015.
378. Judd, J. S. (1989). Neural Network Design and the Complexity of Learning. MIT Press.
379. Jutten, C. and Herault, J. (1991). Blind separation of sources, part I: an adaptive algorithm based on neuromimetic architecture. Signal Processing, 24, 1–10.
380. Kahou, S. E., Pal, C., Bouthillier, X., Froumenty, P., Gülçehre, C., Memisevic, R., Vincent, P., Courville, A., Bengio, Y., Ferrari, R. C., Mirza, M., Jean, S., Carrier, P. L., Dauphin, Y., Boulanger-Lewandowski, N., Aggarwal, A., Zumer, J., Lamblin, P., Raymond, J.-P., Desjardins, G., Pascanu, R., Warde-Farley, D., Torabi, A., Sharma, A., Bengio, E., Côté, M., Konda, K. R., and Wu, Z. (2013). Combining modality specific deep neural networks for emotion recognition in video. In Proceedings of the 15th ACM on International Conference on Multimodal Interaction.
381. Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In EMNLP'2013.
382. Kalchbrenner, N., Danihelka, I., and Graves, A. (2015). Grid long short-term memory. arXiv preprint arXiv:1507.01526.
383. Kamyshanska, H. and Memisevic, R. (2015). The potential energy of an autoencoder. IEEE Transactions on Pattern Analysis and Machine Intelligence.
384. Karpathy, A. and Li, F.-F. (2015). Deep visual-semantic alignments for generating image descriptions. In CVPR'2015. arXiv:1412.2306.
385. Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In CVPR.
386. Karush, W. (1939). Minima of Functions of Several Variables with Inequalities as Side Constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago.
387. Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-35(3), 400–401.
388. Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2008). Fast inference in sparse coding algorithms with applications to object recognition. Technical report, Computational and Biological Learning Lab, Courant Institute, NYU. Tech Report CBLL-TR-2008-12-01.
389. Kavukcuoglu, K., Ranzato, M.-A., Fergus, R., and LeCun, Y. (2009). Learning invariant features through topographic filter maps. In CVPR'2009.
390. Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., and LeCun, Y. (2010). Learning convolutional feature hierarchies for visual recognition. In NIPS'2010.

391. Kelley, H. J. (1960). Gradient theory of optimal flight paths. ARS Journal, 30(10), 947–954.
392. Khan, F., Zhu, X., and Mutlu, B. (2011). How do humans teach: On curriculum learning and teaching dimension. In Advances in Neural Information Processing Systems 24 (NIPS'11), pages 1449–1457.
393. Kim, S. K., McAfee, L. C., McMahon, P. L., and Olukotun, K. (2009). A highly scalable restricted Boltzmann machine FPGA implementation. In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pages 367–372. IEEE.
394. Kindermann, R. (1980). Markov Random Fields and Their Applications (Contemporary Mathematics ; V. 1). American Mathematical Society.
395. Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
396. Kingma, D. and LeCun, Y. (2010). Regularized estimation of image statistics by score matching. In NIPS'2010.
397. Kingma, D., Rezende, D., Mohamed, S., and Welling, M. (2014). Semi-supervised learning with deep generative models. In NIPS'2014.
398. Kingma, D. P. (2013). Fast gradient-based inference with continuous latent variable models in auxiliary form. Technical report, arxiv:1306.0733.
399. Kingma, D. P. and Welling, M. (2014a). Auto-encoding variational bayes. In Proceedings of the International Conference on Learning Representations (ICLR).
400. Kingma, D. P. and Welling, M. (2014b). Efficient gradient-based inference through transformations between bayes nets and neural nets. Technical report, arxiv:1402.0480.
401. Kirkpatrick, S., Jr., C. D. G., , and Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220, 671–680.
402. Kiros, R., Salakhutdinov, R., and Zemel, R. (2014a). Multimodal neural language models. In ICML'2014.
403. Kiros, R., Salakhutdinov, R., and Zemel, R. (2014b). Unifying visual-semantic embeddings with multimodal neural language models. arXiv:1411.2539 [cs.LG].
404. Klementiev, A., Titov, I., and Bhattarai, B. (2012). Inducing crosslingual distributed representations of words. In Proceedings of COLING 2012.
405. Knowles-Barley, S., Jones, T. R., Morgan, J., Lee, D., Kasthuri, N., Lichtman, J. W., and Pfister, H. (2014). Deep learning for the connectome. GPU Technology Conference.
406. Koller, D. and Friedman, N. (2009). Probabilistic Graphical Models: Principles and Techniques. MIT Press.
407. Konig, Y., Bourlard, H., and Morgan, N. (1996). REMAP: Recursive estimation and maximization of a posteriori probabilities – application to transition-based connectionist speech recognition. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, Advances in Neural Information Processing Systems 8 (NIPS'95). MIT Press, Cambridge, MA.
408. Koren, Y. (2009). The BellKor solution to the Netflix grand prize.
409. Kotzias, D., Denil, M., de Freitas, N., and Smyth, P. (2015). From group to individual labels using deep features. In ACM SIGKDD.
410. Koutnik, J., Greff, K., Gomez, F., and Schmidhuber, J. (2014). A clockwork RNN. In ICML'2014.

411. Kociský, T., Hermann, K. M., and Blunsom, P. (2014). Learning Bilingual Word Representations by Marginalizing Alignments. In Proceedings of ACL.
412. Krause, O., Fischer, A., Glasmachers, T., and Igel, C. (2013). Approximation properties of DBNs with binary hidden units and real-valued visible units. In ICML'2013.
413. Krizhevsky, A. (2010). Convolutional deep belief networks on CIFAR-10. Technical report, University of Toronto. Unpublished Manuscript: http://www.cs.utoronto.ca/ kriz/convcifar10-aug2010.pdf.
414. Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.
415. Krizhevsky, A. and Hinton, G. E. (2011). Using very deep autoencoders for content-based image retrieval. In ESANN.
416. Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. In NIPS'2012.
417. Krueger, K. A. and Dayan, P. (2009). Flexible shaping: how learning in small steps helps. Cognition, 110, 380–394.
418. Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear programming. In Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, pages 481–492, Berkeley, Calif. University of California Press.
419. Kumar, A., Irsoy, O., Su, J., Bradbury, J., English, R., Pierce, B., Ondruska, P., Iyyer, M., Gulrajani, I., and Socher, R. (2015). Ask me anything: Dynamic memory networks for natural language processing. arXiv:1506.07285.
420. Kumar, M. P., Packer, B., and Koller, D. (2010). Self-paced learning for latent variable models. In NIPS'2010.
421. Lang, K. J. and Hinton, G. E. (1988). The development of the time-delay neural network architecture for speech recognition. Technical Report CMU-CS-88-152, Carnegie-Mellon University.
422. Lang, K. J., Waibel, A. H., and Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. Neural networks, 3(1), 23–43.
423. Langford, J. and Zhang, T. (2008). The epoch-greedy algorithm for contextual multi-armed bandits. In NIPS'2008, pages 1096–1103.
424. Lappalainen, H., Giannakopoulos, X., Honkela, A., and Karhunen, J. (2000). Nonlinear independent component analysis using ensemble learning: Experiments and discussion. In Proc. ICA. Citeseer.
425. Larochelle, H. and Bengio, Y. (2008). Classification using discriminative restricted Boltzmann machines. In ICML'2008.
426. Larochelle, H. and Hinton, G. E. (2010). Learning to combine foveal glimpses with a third-order Boltzmann machine. In Advances in Neural Information Processing Systems 23, pages 1243–1251.
427. Larochelle, H. and Murray, I. (2011). The Neural Autoregressive Distribution Estimator. In AISTATS'2011.
428. Larochelle, H., Erhan, D., and Bengio, Y. (2008). Zero-data learning of new tasks. In AAAI Conference on Artificial Intelligence.
429. Larochelle, H., Bengio, Y., Louradour, J., and Lamblin, P. (2009). Exploring strategies for training deep neural networks. Journal of Machine Learning Research, 10, 1–40.
430. Lasserre, J. A., Bishop, C. M., and Minka, T. P. (2006). Principled hybrids of generative and discriminative models. In Proceedings of the Computer Vision and Pattern

Recognition Conference (CVPR'06), pages 87–94, Washington, DC, USA. IEEE Computer Society.
431. Le, Q., Ngiam, J., Chen, Z., hao Chia, D. J., Koh, P. W., and Ng, A. (2010). Tiled convolutional neural networks. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, Advances in Neural Information Processing Systems 23 (NIPS'10), pages 1279–1287.
432. Le, Q., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., and Ng, A. (2011). On optimization methods for deep learning. In Proc. ICML'2011. ACM.
433. Le, Q., Ranzato, M., Monga, R., Devin, M., Corrado, G., Chen, K., Dean, J., and Ng, A. (2012). Building high-level features using large scale unsupervised learning. In ICML'2012.
434. Le Roux, N. and Bengio, Y. (2008). Representational power of restricted Boltzmann machines and deep belief networks. Neural Computation, 20(6), 1631–1649.
435. Le Roux, N. and Bengio, Y. (2010). Deep belief networks are compact universal approximators. Neural Computation, 22(8), 2192–2207.
436. LeCun, Y. (1985). Une procédure d'apprentissage pour Réseau à seuil assymétrique. In Cognitiva 85: A la Frontière de l'Intelligence Artificielle, des Sciences de la Connaissance et des Neurosciences, pages 599–604, Paris 1985. CESTA, Paris.
437. LeCun, Y. (1986). Learning processes in an asymmetric threshold network. In F. Fogelman-Soulié, E. Bienenstock, and G. Weisbuch, editors, Disordered Systems and Biological Organization, pages 233–240. Springer-Verlag, Les Houches, France.
438. LeCun, Y. (1987). Modèles connexionistes de l'apprentissage. Ph.D. thesis, Université de Paris VI.
439. LeCun, Y. (1989). Generalization and network design strategies. Technical Report CRG-TR-89-4, University of Toronto.
440. LeCun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., and Hubbard, W. (1989). Handwritten digit recognition: Applications of neural network chips and automatic learning. IEEE Communications Magazine, 27(11), 41–46.
441. LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998a). Efficient backprop. In Neural Networks, Tricks of the Trade, Lecture Notes in Computer Science LNCS 1524. Springer Verlag.
442. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998b). Gradient based learning applied to document recognition. Proc. IEEE.
443. LeCun, Y., Kavukcuoglu, K., and Farabet, C. (2010). Convolutional networks and applications in vision. In Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on, pages 253–256. IEEE.
444. L'Ecuyer, P. (1994). Efficiency improvement and variance reduction. In Proceedings of the 1994 Winter Simulation Conference, pages 122–132.
445. Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. (2014). Deeply-supervised nets. arXiv preprint arXiv:1409.5185.
446. Lee, H., Battle, A., Raina, R., and Ng, A. (2007). Efficient sparse coding algorithms. In B. Schölkopf, J. Platt, and T. Hoffman, editors, Advances in Neural Information Processing Systems 19 (NIPS'06), pages 801–808. MIT Press.
447. Lee, H., Ekanadham, C., and Ng, A. (2008). Sparse deep belief net model for visual area V2. In NIPS'07.

448. Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In L. Bottou and M. Littman, editors, Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09). ACM, Montreal, Canada.
449. Lee, Y. J. and Grauman, K. (2011). Learning the easy things first: self-paced visual category discovery. In CVPR'2011.
450. Leibniz, G. W. (1676). Memoir using the chain rule. (Cited in TMME 7:2&3 p 321–332, 2010).
451. Lenat, D. B. and Guha, R. V. (1989). Building large knowledge-based systems; representation and inference in the Cyc project. Addison-Wesley Longman Publishing Co., Inc.
452. Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks, 6, 861–867.
453. Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. Quarterly Journal of Applied Mathematics, II(2), 164–168.
454. L'Hôpital, G. F. A. (1696). Analyse des infiniment petits, pour l'intelligence des lignes courbes. Paris: L'Imprimerie Royale.
455. Li, Y., Swersky, K., and Zemel, R. S. (2015). Generative moment matching networks. CoRR, abs/1502.02761.
456. Lin, T., Horne, B. G., Tino, P., and Giles, C. L. (1996). Learning long-term dependencies is not as difficult with NARX recurrent neural networks. IEEE Transactions on Neural Networks, 7(6), 1329–1338.
457. Lin, Y., Liu, Z., Sun, M., Liu, Y., and Zhu, X. (2015). Learning entity and relation embeddings for knowledge graph completion. In Proc. AAAI'15.
458. Linde, N. (1992). The machine that changed the world, episode 3. Documentary miniseries.
459. Lindsey, C. and Lindblad, T. (1994). Review of hardware neural networks: a user's perspective. In Proc. Third Workshop on Neural Networks: From Biology to High Energy Physics, pages 195–202, Isola d'Elba, Italy.
460. Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. BIT Numerical Mathematics, 16(2), 146–160.
461. LISA (2008). Deep learning tutorials: Restricted Boltzmann machines. Technical report, LISA Lab, Université de Montréal.
462. Long, P. M. and Servedio, R. A. (2010). Restricted Boltzmann machines are hard to approximately evaluate or simulate. In Proceedings of the 27th International Conference on Machine Learning (ICML'10).
463. Lotter, W., Kreiman, G., and Cox, D. (2015). Unsupervised learning of visual structure using predictive generative networks. arXiv preprint arXiv:1511.06380.
464. Lovelace, A. (1842). Notes upon L. F. Menabrea's «Sketch of the Analytical Engine invented by Charles Babbage».
465. Lu, L., Zhang, X., Cho, K., and Renals, S. (2015). A study of the recurrent neural network encoder-decoder for large vocabulary speech recognition. In Proc. Interspeech.
466. Lu, T., Pál, D., and Pál, M. (2010). Contextual multi-armed bandits. In International Conference on Artificial Intelligence and Statistics, pages 485–492.
467. Luenberger, D. G. (1984). Linear and Nonlinear Programming. Addison Wesley.

468. Lukoševicius, M. and Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. Computer Science Review, 3(3), 127–149.
469. Luo, H., Shen, R., Niu, C., and Ullrich, C. (2011). Learning class-relevant features and class-irrelevant features via a hybrid third-order RBM. In International Conference on Artificial Intelligence and Statistics, pages 470–478.
470. Luo, H., Carrier, P. L., Courville, A., and Bengio, Y. (2013). Texture modeling with convolutional spike-and-slab RBMs and deep extensions. In AISTATS'2013.
471. Lyu, S. (2009). Interpretation and generalization of score matching. In Proceedings of the Twenty-fifth Conference in Uncertainty in Artificial Intelligence (UAI'09).
472. Ma, J., Sheridan, R. P., Liaw, A., Dahl, G. E., and Svetnik, V. (2015). Deep neural nets as a method for quantitative structure – activity relationships. J. Chemical information and modeling.
473. Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In ICML Workshop on Deep Learning for Audio, Speech, and Language Processing.
474. Maass, W. (1992). Bounds for the computational power and learning complexity of analog neural nets (extended abstract). In Proc. of the 25th ACM Symp. Theory of Computing, pages 335–344.
475. Maass, W., Schnitger, G., and Sontag, E. D. (1994). A comparison of the computational power of sigmoid and Boolean threshold circuits. Theoretical Advances in Neural Computation and Learning, pages 127–151.
476. Maass, W., Natschlaeger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural Computation, 14(11), 2531–2560.
477. MacKay, D. (2003). Information Theory, Inference and Learning Algorithms. Cambridge University Press.
478. Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. arXiv preprint arXiv:1502.03492.
479. Mao, J., Xu, W., Yang, Y., Wang, J., Huang, Z., and Yuille, A. L. (2015). Deep captioning with multimodal recurrent neural networks. In ICLR'2015. arXiv:1410.1090.
480. Marcotte, P. and Savard, G. (1992). Novel approaches to the discrimination problem. Zeitschrift für Operations Research (Theory), 36, 517–545.
481. Marlin, B. and de Freitas, N. (2011). Asymptotic efficiency of deterministic estimators for discrete energy-based models: Ratio matching and pseudolikelihood. In UAI'2011.
482. Marlin, B., Swersky, K., Chen, B., and de Freitas, N. (2010). Inductive principles for restricted Boltzmann machine learning. In Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10), volume 9, pages 509–516.
483. Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. Journal of the Society of Industrial and Applied Mathematics, 11(2), 431–441.
484. Marr, D. and Poggio, T. (1976). Cooperative computation of stereo disparity. Science, 194.
485. Martens, J. (2010). Deep learning via Hessian-free optimization. In L. Bottou and M. Littman, editors, Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10), pages 735–742. ACM.

486. Martens, J. and Medabalimi, V. (2014). On the expressive efficiency of sum product networks. arXiv:1411.7717.
487. Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with Hessian-free optimization. In Proc. ICML'2011. ACM.
488. Mase, S. (1995). Consistency of the maximum pseudo-likelihood estimator of continuous state space Gibbsian processes. The Annals of Applied Probability, 5(3), pp. 603–612.
489. McClelland, J., Rumelhart, D., and Hinton, G. (1995). The appeal of parallel distributed processing. In Computation & intelligence, pages 305–341. American Association for Artificial Intelligence.
490. McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5, 115–133.
491. Mead, C. and Ismail, M. (2012). Analog VLSI implementation of neural systems, volume 80. Springer Science & Business Media.
492. Melchior, J., Fischer, A., and Wiskott, L. (2013). How to center binary deep Boltzmann machines. arXiv preprint arXiv:1311.1354.
493. Memisevic, R. and Hinton, G. E. (2007). Unsupervised learning of image transformations. In Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07).
494. Memisevic, R. and Hinton, G. E. (2010). Learning to represent spatial transformations with factored higher-order Boltzmann machines. Neural Computation, 22(6), 1473–1492.
495. Mesnil, G., Dauphin, Y., Glorot, X., Rifai, S., Bengio, Y., Goodfellow, I., Lavoie, E., Muller, X., Desjardins, G., Warde-Farley, D., Vincent, P., Courville, A., and Bergstra, J. (2011). Unsupervised and transfer learning challenge: a deep learning approach. In JMLR W&CP: Proc. Unsupervised and Transfer Learning, volume 7.
496. Mesnil, G., Rifai, S., Dauphin, Y., Bengio, Y., and Vincent, P. (2012). Surfing on the manifold. Learning Workshop, Snowbird.
497. Miikkulainen, R. and Dyer, M. G. (1991). Natural language processing with modular PDP networks and distributed lexicon. Cognitive Science, 15, 343–399.
498. Mikolov, T. (2012). Statistical Language Models based on Neural Networks. Ph. D. thesis, Brno University of Technology.
499. Mikolov, T., Deoras, A., Kombrink, S., Burget, L., and Cernocky, J. (2011a). Empirical evaluation and combination of advanced language modeling techniques. In Proc. 12th annual conference of the international speech communication association (INTERSPEECH 2011).
500. Mikolov, T., Deoras, A., Povey, D., Burget, L., and Cernocky, J. (2011b). Strategies for training large scale neural network language models. In Proc. ASRU'2011.
501. Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. In International Conference on Learning Representations: Workshops Track.
502. Mikolov, T., Le, Q. V., and Sutskever, I. (2013b). Exploiting similarities among languages for machine translation. Technical report, arXiv:1309.4168.
503. Minka, T. (2005). Divergence measures and message passing. Microsoft Research Cambridge UK Tech Rep MSRTR2005173, 72(TR-2005-173).
504. Minsky, M. L. and Papert, S. A. (1969). Perceptrons. MIT Press, Cambridge.

505. Mirza, M. and Osindero, S. (2014). Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784.
506. Mishkin, D. and Matas, J. (2015). All you need is a good init. arXiv preprint arXiv:1511.06422.
507. Misra, J. and Saha, I. (2010). Artificial neural networks in hardware: A survey of two decades of progress. Neurocomputing, 74(1), 239–255.
508. Mitchell, T. M. (1997). Machine Learning. McGraw-Hill, New York.
509. Miyato, T., Maeda, S., Koyama, M., Nakae, K., and Ishii, S. (2015). Distributional smoothing with virtual adversarial training. In ICLR. Preprint: arXiv:1507.00677.
510. Mnih, A. and Gregor, K. (2014). Neural variational inference and learning in belief networks. In ICML'2014.
511. Mnih, A. and Hinton, G. E. (2007). Three new graphical models for statistical language modelling. In Z. Ghahramani, editor, Proceedings of the Twenty-fourth International Conference on Machine Learning (ICML'07), pages 641–648. ACM.
512. Mnih, A. and Hinton, G. E. (2009). A scalable hierarchical distributed language model. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, Advances in Neural Information Processing Systems 21 (NIPS'08), pages 1081–1088.
513. Mnih, A. and Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noisecontrastive estimation. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 2265–2273. Curran Associates, Inc.
514. Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. In ICML'2012, pages 1751–1758.
515. Mnih, V. and Hinton, G. (2010). Learning to detect roads in high-resolution aerial images. In Proceedings of the 11th European Conference on Computer Vision (ECCV).
516. Mnih, V., Larochelle, H., and Hinton, G. (2011). Conditional restricted Boltzmann machines for structure output prediction. In Proc. Conf. on Uncertainty in Artificial Intelligence (UAI).
517. Mnih, V., Kavukcuoglo, K., Silver, D., Graves, A., Antonoglou, I., and Wierstra, D. (2013). Playing Atari with deep reinforcement learning. Technical report, arXiv: 1312.5602.
518. Mnih, V., Heess, N., Graves, A., and Kavukcuoglu, K. (2014). Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, NIPS'2014, pages 2204–2212.
519. Mnih, V., Kavukcuoglo, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidgeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529–533.
520. Mobahi, H. and Fisher, III, J. W. (2015). A theoretical analysis of optimization by Gaussian continuation. In AAAI'2015.
521. Mobahi, H., Collobert, R., and Weston, J. (2009). Deep learning from temporal coherence in video. In L. Bottou and M. Littman, editors, Proceedings of the 26th International Conference on Machine Learning, pages 737–744, Montreal. Omnipress.
522. Mohamed, A., Dahl, G., and Hinton, G. (2009). Deep belief networks for phone recognition.

523. Mohamed, A., Sainath, T. N., Dahl, G., Ramabhadran, B., Hinton, G. E., and Pichen y, M. A. (2011). Deep belief networks using discriminative features for phone recognition. In Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, pages 5060–5063. IEEE.
524. Mohamed, A., Dahl, G., and Hinton, G. (2012a). Acoustic modeling using deep belief networks. IEEE Trans. on Audio, Speech and Language Processing, 20(1), 14–22.
525. Mohamed, A., Hinton, G., and Penn, G. (2012b). Understanding how deep belief networks perform acoustic modelling. In Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on, pages 4273–4276. IEEE.
526. Moller, M. F. (1993). A scaled conjugate gradient algorithm for fast supervised learning. Neural Networks, 6, 525–533.
527. Montavon, G. and Muller, K.-R. (2012). Deep Boltzmann machines and the centering trick. In G. Montavon, G. Orr, and K.-R. Müller, editors, Neural Networks: Tricks of the Trade, volume 7700 of Lecture Notes in Computer Science, pages 621–637. Preprint: http://arxiv.org/abs/1203.3783.
528. Montúfar, G. (2014). Universal approximation depth and errors of narrow belief networks with discrete units. Neural Computation, 26.
529. Montúfar, G. and Ay, N. (2011). Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. Neural Computation, 23(5), 1306–1319.
530. Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. (2014). On the number of linear regions of deep neural networks. In NIPS'2014.
531. Mor-Yosef, S., Samueloff, A., Modan, B., Navot, D., and Schenker, J. G. (1990). Ranking the risk factors for cesarean: logistic regression analysis of a nationwide study. Obstet Gynecol, 75(6), 944–7.
532. Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In AISTATS'2005.
533. Mozer, M. C. (1992). The induction of multiscale temporal structure. In J. M. S. Hanson and R. Lippmann, editors, Advances in Neural Information Processing Systems 4 (NIPS'91), pages 275–282, San Mateo, CA. Morgan Kaufmann.
534. Murphy, K. P. (2012). Machine Learning: a Probabilistic Perspective. MIT Press, Cambridge, MA, USA.
535. Murray, B. U. I. and Larochelle, H. (2014). A deep and tractable density estimator. In ICML'2014.
536. Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. In ICML'2010.
537. Nair, V. and Hinton, G. E. (2009). 3d object recognition with deep belief nets. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, Advances in Neural Information Processing Systems 22, pages 1339–1347. Curran Associates, Inc.
538. Narayanan, H. and Mitter, S. (2010). Sample complexity of testing the manifold hypothesis. In NIPS'2010.
539. Naumann, U. (2008). Optimal Jacobian accumulation is NP-complete. Mathematical Programming, 112(2), 427–441.
540. Navigli, R. and Velardi, P. (2005). Structural semantic interconnections: a knowledge-based approach to word sense disambiguation. IEEE Trans. Pattern Analysis and Machine Intelligence, 27(7), 1075–1086.

541. Neal, R. and Hinton, G. (1999). A view of the EM algorithm that justifies incremental, sparse, and other variants. In M. I. Jordan, editor, Learning in Graphical Models. MIT Press, Cambridge, MA.
542. Neal, R. M. (1990). Learning stochastic feedforward networks. Technical report.
543. Neal, R. M. (1993). Probabilistic inference using Markov chain Monte-Carlo methods. Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto.
544. Neal, R. M. (1994). Sampling from multimodal distributions using tempered transitions. Technical Report 9421, Dept. of Statistics, University of Toronto.
545. Neal, R. M. (1996). Bayesian Learning for Neural Networks. Lecture Notes in Statistics. Springer.
546. Neal, R. M. (2001). Annealed importance sampling. Statistics and Computing, 11(2), 125–139.
547. Neal, R. M. (2005). Estimating ratios of normalizing constants using linked importance sampling.
548. Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate O(1/k2). Soviet Mathematics Doklady, 27, 372–376.
549. Nesterov, Y. (2004). Introductory lectures on convex optimization : a basic course. Applied optimization. Kluwer Academic Publ., Boston, Dordrecht, London.
550. Netzer, Y.,Wang, T., Coates, A., Bissacco, A.,Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. Deep Learning and Unsupervised Feature Learning Workshop, NIPS.
551. Ney, H. and Kneser, R. (1993). Improved clustering techniques for class-based statistical language modelling. In European Conference on Speech Communication and Technology (Eurospeech), pages 973–976, Berlin.
552. Ng, A. (2015). Advice for applying machine learning. https://see.stanford.edu/materials/aimlcs229/ML-advice.pdf.
553. Niesler, T. R., Whittaker, E. W. D., and Woodland, P. C. (1998). Comparison of part-ofspeech and automatically derived category-based language models for speech recognition. In International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 177–180.
554. Ning, F., Delhomme, D., LeCun, Y., Piano, F., Bottou, L., and Barbano, P. E. (2005). Toward automatic phenotyping of developing embryos from videos. Image Processing, IEEE Transactions on, 14(9), 1360–1371.
555. Nocedal, J. and Wright, S. (2006). Numerical Optimization. Springer.
556. Norouzi, M. and Fleet, D. J. (2011). Minimal loss hashing for compact binary codes. In ICML'2011.
557. Nowlan, S. J. (1990). Competing experts: An experimental investigation of associative mixture models. Technical Report CRG-TR-90-5, University of Toronto.
558. Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. Neural Computation, 4(4), 473–493.
559. Olshausen, B. and Field, D. J. (2005). How close are we to understanding V1? Neural Computation, 17, 1665–1699.
560. Olshausen, B. A. and Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature, 381, 607–609.
561. Olshausen, B. A., Anderson, C. H., and Van Essen, D. C. (1993). A neurobiological model of visual attention and invariant pattern recognition based on dynamic routing of information. J. Neurosci., 13(11), 4700–4719.

562. Opper, M. and Archambeau, C. (2009). The variational Gaussian approximation revisited. Neural computation, 21(3), 786–792.
563. Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, pages 1717–1724. IEEE.
564. Osindero, S. and Hinton, G. E. (2008). Modeling image patches with a directed hierarchy of Markov random fields. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, Advances in Neural Information Processing Systems 20 (NIPS'07), pages 1121–1128, Cambridge, MA. MIT Press.
565. Ovid and Martin, C. (2004). Metamorphoses. W.W. Norton.
566. Paccanaro, A. and Hinton, G. E. (2000). Extracting distributed representations of concepts and relations from positive and negative propositions. In International Joint Conference on Neural Networks (IJCNN), Como, Italy. IEEE, New York.
567. Paine, T. L., Khorrami, P., Han, W., and Huang, T. S. (2014). An analysis of unsupervised pre-training in light of recent advances. arXiv preprint arXiv:1412.6597.
568. Palatucci, M., Pomerleau, D., Hinton, G. E., and Mitchell, T. M. (2009). Zero-shot learning with semantic output codes. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, Advances in Neural Information Processing Systems 22, pages 1410–1418. Curran Associates, Inc.
569. Parker, D. B. (1985). Learning-logic. Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT.
570. Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In ICML'2013.
571. Pascanu, R., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014a). How to construct deep recurrent neural networks. In ICLR'2014.
572. Pascanu, R., Montufar, G., and Bengio, Y. (2014b). On the number of inference regions of deep feed forward networks with piece-wise linear activations. In ICLR'2014.
573. Pati, Y., Rezaiifar, R., and Krishnaprasad, P. (1993). Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In Proceedings of the 27 th Annual Asilomar Conference on Signals, Systems, and Computers, pages 40–44.
574. Pearl, J. (1985). Bayesian networks: A model of self-activated memory for evidential reasoning. In Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, pages 329–334.
575. Pearl, J. (1988). Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann.
576. Perron, O. (1907). Zur theorie der matrices. Mathematische Annalen, 64(2), 248–263.
577. Petersen, K. B. and Pedersen, M. S. (2006). The matrix cookbook. Version 20051003.
578. Peterson, G. B. (2004). A day of great illumination: B. F. Skinner's discovery of shaping. Journal of the Experimental Analysis of Behavior, 82(3), 317–328.
579. Pham, D.-T., Garat, P., and Jutten, C. (1992). Separation of a mixture of independent sources through a maximum likelihood approach. In EUSIPCO, pages 771–774.
580. Pham, P.-H., Jelaca, D., Farabet, C., Martini, B., LeCun, Y., and Culurciello, E. (2012). NeuFlow: dataflow vision processing system-on-a-chip. In Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on, pages 1044–1047. IEEE.

581. Pinheiro, P. H. O. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In ICML'2014.
582. Pinheiro, P. H. O. and Collobert, R. (2015). From image-level to pixel-level labeling with convolutional networks. In Conference on Computer Vision and Pattern Recognition (CVPR).
583. Pinto, N., Cox, D. D., and DiCarlo, J. J. (2008). Why is real-world visual object recognition hard? PLoS Comput Biol, 4.
584. Pinto, N., Stone, Z., Zickler, T., and Cox, D. (2011). Scaling up biologically-inspired computer vision: A case study in unconstrained face recognition on facebook. In Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on, pages 35–42. IEEE.
585. Pollack, J. B. (1990). Recursive distributed representations. Artificial Intelligence, 46(1), 77–105.
586. Polyak, B. and Juditsky, A. (1992). Acceleration of stochastic approximation by averaging. SIAM J. Control and Optimization, 30(4), 838–855.
587. Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics, 4(5), 1–17.
588. Poole, B., Sohl-Dickstein, J., and Ganguli, S. (2014). Analyzing noise in autoencoders and deep networks. CoRR, abs/1406.1831.
589. Poon, H. and Domingos, P. (2011). Sum-product networks: A new deep architecture. In Proceedings of the Twenty-seventh Conference in Uncertainty in Artificial Intelligence (UAI), Barcelona, Spain.
590. Presley, R. K. and Haggard, R. L. (1994). A fixed point implementation of the backpropagation learning algorithm. In Southeastcon'94. Creative Technology Transfer-A Global Affair., Proceedings of the 1994 IEEE, pages 136–138. IEEE.
591. Price, R. (1958). A useful theorem for nonlinear devices having Gaussian inputs. IEEE Transactions on Information Theory, 4(2), 69–72.
592. Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., and Fried, I. (2005). Invariant visual representation by single neurons in the human brain. Nature, 435(7045), 1102–1107.
593. Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv: 1511.06434.
594. Raiko, T., Yao, L., Cho, K., and Bengio, Y. (2014). Iterative neural autoregressive distribution estimator (NADE-k). Technical report, arXiv:1406.1485.
595. Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In L. Bottou and M. Littman, editors, Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09), pages 873–880, New York, NY, USA. ACM.
596. Ramsey, F. P. (1926). Truth and probability. In R. B. Braithwaite, editor, The Foundations of Mathematics and other Logical Essays, chapter 7, pages 156–198. McMaster University Archive for the History of Economic Thought.
597. Ranzato, M. and Hinton, G. H. (2010). Modeling pixel means and covariances using factorized third-order Boltzmann machines. In CVPR'2010, pages 2551–2558.
598. Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007a). Efficient learning of sparse representations with an energy-based model. In NIPS'2006.
599. Ranzato, M., Huang, F., Boureau, Y., and LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In Pro-

ceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07). IEEE Press.
600. Ranzato, M., Boureau, Y., and LeCun, Y. (2008). Sparse feature learning for deep belief networks. In NIPS'2007.
601. Ranzato, M., Krizhevsky, A., and Hinton, G. E. (2010a). Factored 3-way restricted Boltzmann machines for modeling natural images. In Proceedings of AISTATS 2010.
602. Ranzato, M., Mnih, V., and Hinton, G. (2010b). Generating more realistic images using gated MRFs. In NIPS'2010.
603. Rao, C. (1945). Information and the accuracy attainable in the estimation of statistical parameters. Bulletin of the Calcutta Mathematical Society, 37, 81–89.
604. Rasmus, A., Valpola, H., Honkala, M., Berglund, M., and Raiko, T. (2015). Semi-supervised learning with ladder network. arXiv preprint arXiv:1507.02672.
605. Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In NIPS'2011.
606. Reichert, D. P., Seriès, P., and Storkey, A. J. (2011). Neuronal adaptation for sampling based probabilistic inference in perceptual bistability. In Advances in Neural Information Processing Systems, pages 2357–2365.
607. Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In ICML'2014. Preprint: arXiv:1401.4082.
608. Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011a). Contractive auto-encoders: Explicit invariance during feature extraction. In ICML'2011.
609. Rifai, S., Mesnil, G., Vincent, P., Muller, X., Bengio, Y., Dauphin, Y., and Glorot, X. (2011b). Higher order contractive auto-encoder. In ECML PKDD.
610. Rifai, S., Dauphin, Y., Vincent, P., Bengio, Y., and Muller, X. (2011c). The manifold tangent classifier. In NIPS'2011.
611. Rifai, S., Bengio, Y., Dauphin, Y., and Vincent, P. (2012). A generative process for sampling contractive auto-encoders. In ICML'2012.
612. Ringach, D. and Shapley, R. (2004). Reverse correlation in neurophysiology. Cognitive Science, 28(2), 147–166.
613. Roberts, S. and Everson, R. (2001). Independent component analysis: principles and practice. Cambridge University Press.
614. Robinson, A. J. and Fallside, F. (1991). A recurrent error propagation network speech recognition system. Computer Speech and Language, 5(3), 259–274.
615. Rockafellar, R. T. (1997). Convex analysis. princeton landmarks in mathematics.
616. Romero, A., Ballas, N., Ebrahimi Kahou, S., Chassang, A., Gatta, C., and Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In ICLR'2015, arXiv:1412.6550.
617. Rosen, J. B. (1960). The gradient projection method for nonlinear programming. Part I. Linear constraints. Journal of the Society for Industrial and Applied Mathematics, 8(1), pp. 181–217.
618. Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, b, 386–408.
619. Rosenblatt, F. (1962). Principles of Neurodynamics. Spartan, New York.
620. Roweis, S. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. Science, 290(5500).
621. Roweis, S., Saul, L., and Hinton, G. (2002). Global coordination of local linear models. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, Advances in Neural Information Processing Systems 14 (NIPS'01), Cambridge, MA. MIT Press.

622. Rubin, D. B. et al. (1984). Bayesianly justifiable and relevant frequency calculations for the applied statistician. The Annals of Statistics, 12(4), 1151–1172.
623. Rumelhart, D., Hinton, G., and Williams, R. (1986a). Learning representations by back-propagating errors. Nature, 323, 533–536.
624. Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge.
625. Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986c). Parallel Distributed Processing: Explorations in the Microstructure of Cognition. MIT Press, Cambridge.
626. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014a). ImageNet Large Scale Visual Recognition Challenge.
627. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2014b). Imagenet large scale visual recognition challenge. arXiv preprint arXiv:1409.0575.
628. Russel, S. J. and Norvig, P. (2003). Artificial Intelligence: a Modern Approach. Prentice Hall.
629. Rust, N., Schwartz, O., Movshon, J. A., and Simoncelli, E. (2005). Spatiotemporal elements of macaque V1 receptive fields. Neuron, 46(6), 945–956.
630. Sainath, T., Mohamed, A., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for LVCSR. In ICASSP 2013.
631. Salakhutdinov, R. (2010). Learning in Markov random fields using tempered transitions. In Y. Bengio, D. Schuurmans, C. Williams, J. Lafferty, and A. Culotta, editors, Advances in Neural Information Processing Systems 22 (NIPS'09).
632. Salakhutdinov, R. and Hinton, G. (2009a). Deep Boltzmann machines. In Proceedings of the International Conference on Artificial Intelligence and Statistics, volume 5, pages 448–455.
633. Salakhutdinov, R. and Hinton, G. (2009b). Semantic hashing. In International Journal of Approximate Reasoning.
634. Salakhutdinov, R. and Hinton, G. E. (2007a). Learning a nonlinear embedding by preserving class neighbourhood structure. In Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS'07), San Juan, Porto Rico. Omnipress.
635. Salakhutdinov, R. and Hinton, G. E. (2007b). Semantic hashing. In SIGIR'2007.
636. Salakhutdinov, R. and Hinton, G. E. (2008). Using deep belief nets to learn covariance kernels for Gaussian processes. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, Advances in Neural Information Processing Systems 20 (NIPS'07), pages 1249–1256, Cambridge, MA. MIT Press.
637. Salakhutdinov, R. and Larochelle, H. (2010). Efficient learning of deep Boltzmann machines. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010), JMLR W&CP, volume 9, pages 693–700.
638. Salakhutdinov, R. and Mnih, A. (2008). Probabilistic matrix factorization. In NIPS'2008.
639. Salakhutdinov, R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, Proceedings of

the Twenty-fifth International Conference on Machine Learning (ICML'08), volume 25, pages 872–879. ACM.
640. Salakhutdinov, R., Mnih, A., and Hinton, G. (2007). Restricted Boltzmann machines for collaborative filtering. In ICML.
641. Sanger, T. D. (1994). Neural network learning control of robot manipulators using gradually increasing task difficulty. IEEE Transactions on Robotics and Automation, 10(3).
642. Saul, L. K. and Jordan, M. I. (1996). Exploiting tractable substructures in intractable networks. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, Advances in Neural Information Processing Systems 8 (NIPS'95). MIT Press, Cambridge, MA.
643. Saul, L. K., Jaakkola, T., and Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. Journal of Artificial Intelligence Research, 4, 61–76.
644. Savich, A. W., Moussa, M., and Areibi, S. (2007). The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. Neural Networks, IEEE Transactions on, 18(1), 240–252.
645. Saxe, A. M., Koh, P. W., Chen, Z., Bhand, M., Suresh, B., and Ng, A. (2011). On random weights and unsupervised feature learning. In Proc. ICML'2011. ACM.
646. Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In ICLR.
647. Schaul, T., Antonoglou, I., and Silver, D. (2014). Unit tests for stochastic optimization. In International Conference on Learning Representations.
648. Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. Neural Computation, 4(2), 234–242.
649. Schmidhuber, J. (1996). Sequential neural text compression. IEEE Transactions on Neural Networks, 7(1), 142–146.
650. Schmidhuber, J. (2012). Self-delimiting neural networks. arXiv preprint arXiv:1210.0118.
651. Schölkopf, B. and Smola, A. J. (2002). Learning with kernels: Support vector machines, regularization, optimization, and beyond. MIT Press.
652. Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. Neural Computation, 10, 1299–1319.
653. Schölkopf, B., Burges, C. J. C., and Smola, A. J. (1999). Advances in Kernel Methods – Support Vector Learning. MIT Press, Cambridge, MA.
654. Schölkopf, B., Janzing, D., Peters, J., Sgouritsa, E., Zhang, K., and Mooij, J. (2012). On causal and anticausal learning. In ICML'2012, pages 1255–1262.
655. Schuster, M. (1999). On supervised learning from sequential data with applications for speech recognition.
656. Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673–2681.
657. Schwenk, H. (2007). Continuous space language models. Computer speech and language, 21, 492–518.
658. Schwenk, H. (2010). Continuous space language models for statistical machine translation. The Prague Bulletin of Mathematical Linguistics, 93, 137–146.
659. Schwenk, H. (2014). Cleaned subset of WMT '14 dataset.
660. Schwenk, H. and Bengio, Y. (1998). Training methods for adaptive boosting of neural networks. In M. Jordan, M. Kearns, and S. Solla, editors, Advances in Neural Information Processing Systems 10 (NIPS'97), pages 647–653. MIT Press.

661. Schwenk, H. and Gauvain, J.-L. (2002). Connectionist language modeling for large vocabulary continuous speech recognition. In International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 765–768, Orlando, Florida.
662. Schwenk, H., Costa-jussà, M. R., and Fonollosa, J. A. R. (2006). Continuous space language models for the IWSLT 2006 task. In International Workshop on Spoken Language Translation, pages 166–173.
663. Seide, F., Li, G., and Yu, D. (2011). Conversational speech transcription using contextdependent deep neural networks. In Interspeech 2011, pages 437–440.
664. Sejnowski, T. (1987). Higher-order Boltzmann machines. In AIP Conference Proceedings 151 on Neural Networks for Computing, pages 398–403. American Institute of Physics Inc.
665. Series, P., Reichert, D. P., and Storkey, A. J. (2010). Hallucinations in Charles Bonnet syndrome induced by homeostasis: a deep Boltzmann machine model. In Advances in Neural Information Processing Systems, pages 2020–2028.
666. Sermanet, P., Chintala, S., and LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. CoRR, abs/1204.3968.
667. Sermanet, P., Kavukcuoglu, K., Chintala, S., and LeCun, Y. (2013). Pedestrian detection with unsupervised multi-stage feature learning. In Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'13). IEEE.
668. Shilov, G. (1977). Linear Algebra. Dover Books on Mathematics Series. Dover Publications.
669. Siegelmann, H. (1995). Computation beyond the Turing limit. Science, 268(5210), 545–548.
670. Siegelmann, H. and Sontag, E. (1991). Turing computability with neural nets. Applied Mathematics Letters, 4(6), 77–80.
671. Siegelmann, H. T. and Sontag, E. D. (1995). On the computational power of neural nets. Journal of Computer and Systems Sciences, 50(1), 132–150.
672. Sietsma, J. and Dow, R. (1991). Creating artificial neural networks that generalize. Neural Networks, 4(1), 67–79.
673. Simard, D., Steinkraus, P. Y., and Platt, J. C. (2003). Best practices for convolutional neural networks. In ICDAR'2003.
674. Simard, P. and Graf, H. P. (1994). Backpropagation without multiplication. In Advances in Neural Information Processing Systems, pages 232–239.
675. Simard, P., Victorri, B., LeCun, Y., and Denker, J. (1992). Tangent prop – A formalism for specifying selected invariances in an adaptive network. In NIPS'1991.
676. Simard, P. Y., LeCun, Y., and Denker, J. (1993). Efficient pattern recognition using a new transformation distance. In NIPS'92.
677. Simard, P. Y., LeCun, Y. A., Denker, J. S., and Victorri, B. (1998). Transformation invariance in pattern recognition – tangent distance and tangent propagation. Lecture Notes in Computer Science, 1524.
678. Simons, D. J. and Levin, D. T. (1998). Failure to detect changes to people during a real-world interaction. Psychonomic Bulletin & Review, 5(4), 644–649.
679. Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In ICLR.
680. Sjöberg, J. and Ljung, L. (1995). Overtraining, regularization and searching for a minimum, with application to neural networks. International Journal of Control, 62(6), 1391–1407.

681. Skinner, B. F. (1958). Reinforcement today. American Psychologist, 13, 94–99.
682. Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing, volume 1, chapter 6, pages 194–281. MIT Press, Cambridge.
683. Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In NIPS'2012.
684. Socher, R., Huang, E. H., Pennington, J., Ng, A. Y., and Manning, C. D. (2011a). Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In NIPS'2011.
685. Socher, R., Manning, C., and Ng, A. Y. (2011b). Parsing natural scenes and natural language with recursive neural networks. In Proceedings of the Twenty-Eighth International Conference on Machine Learning (ICML'2011).
686. Socher, R., Pennington, J., Huang, E. H., Ng, A. Y., and Manning, C. D. (2011c). Semi-supervised recursive autoencoders for predicting sentiment distributions. In EMNLP'2011.
687. Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013a). Recursive deep models for semantic compositionality over a sentiment treebank. In EMNLP'2013.
688. Socher, R., Ganjoo, M., Manning, C. D., and Ng, A. Y. (2013b). Zero-shot learning through cross-modal transfer. In 27th Annual Conference on Neural Information Processing Systems (NIPS 2013).
689. Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., and Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics.
690. Sohn, K., Zhou, G., and Lee, H. (2013). Learning and selecting features jointly with point-wise gated Boltzmann machines. In ICML'2013.
691. Solomonoff, R. J. (1989). A system for incremental learning based on algorithmic probability.
692. Sontag, E. D. (1998). VC dimension of neural networks. NATO ASI Series F Computer and Systems Sciences, 168, 69–96.
693. Sontag, E. D. and Sussman, H. J. (1989). Backpropagation can give rise to spurious local minima even for networks without hidden layers. Complex Systems, 3, 91–106.
694. Sparkes, B. (1996). The Red and the Black: Studies in Greek Pottery. Routledge.
695. Spitkovsky, V. I., Alshawi, H., and Jurafsky, D. (2010). From baby steps to leapfrog: how "less is more" in unsupervised dependency parsing. In HLT'10.
696. Squire, W. and Trapp, G. (1998). Using complex variables to estimate derivatives of real functions. SIAM Rev., 40(1), 110–112.
697. Srebro, N. and Shraibman, A. (2005). Rank, trace-norm and max-norm. In Proceedings of the 18th Annual Conference on Learning Theory, pages 545–560. Springer-Verlag.
698. Srivastava, N. (2013). Improving Neural Networks With Dropout. Master's thesis, U. Toronto.
699. Srivastava, N. and Salakhutdinov, R. (2012). Multimodal learning with deep Boltzmann machines. In NIPS'2012.
700. Srivastava, N., Salakhutdinov, R. R., and Hinton, G. E. (2013). Modeling documents with deep Boltzmann machines. arXiv preprint arXiv:1309.6865.

701. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15, 1929–1958.
702. Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Highway networks. arXiv: 1505.00387.
703. Steinkrau, D., Simard, P. Y., and Buck, I. (2005). Using GPUs for machine learning algorithms. 2013 12th International Conference on Document Analysis and Recognition, 0, 1115–1119.
704. Stoyanov, V., Ropson, A., and Eisner, J. (2011). Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure. In Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS), volume 15 of JMLR Workshop and Conference Proceedings, pages 725–733, Fort Lauderdale. Supplementary material (4 pages) also available.
705. Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. (2015). Weakly supervised memory networks. arXiv preprint arXiv:1503.08895.
706. Supancic, J. and Ramanan, D. (2013). Self-paced learning for long-term tracking. In CVPR'2013.
707. Sussillo, D. (2014). Random walks: Training very deep nonlinear feed-forward networks with smart initialization. CoRR, abs/1412.6558.
708. Sutskever, I. (2012). Training Recurrent Neural Networks. Ph.D. thesis, Department of computer science, University of Toronto.
709. Sutskever, I. and Hinton, G. E. (2008). Deep narrow sigmoid belief networks are universal approximators. Neural Computation, 20(11), 2629–2636.
710. Sutskever, I. and Tieleman, T. (2010). On the Convergence Properties of Contrastive Divergence. In Y. W. Teh and M. Titterington, editors, Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS), volume 9, pages 789–795.
711. Sutskever, I., Hinton, G., and Taylor, G. (2009). The recurrent temporal restricted Boltzmann machine. In NIPS'2008.
712. Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In ICML'2011, pages 1017–1024.
713. Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In ICML.
714. Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In NIPS'2014, arXiv:1409.3215.
715. Sutton, R. and Barto, A. (1998). Reinforcement Learning: An Introduction. MIT Press.
716. Sutton, R. S., Mcallester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In NIPS'1999, pages 1057–1063. MIT Press.
717. Swersky, K., Ranzato, M., Buchman, D., Marlin, B., and de Freitas, N. (2011). On autoencoders and score matching for energy based models. In ICML'2011. ACM.
718. Swersky, K., Snoek, J., and Adams, R. P. (2014). Freeze-thaw Bayesian optimization. arXiv preprint arXiv:1406.3896.
719. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014a). Going deeper with convolutions. Technical report, arXiv:1409.4842.

720. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2014b). Intriguing properties of neural networks. ICLR, abs/1312.6199.
721. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision. ArXiv e-prints.
722. Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). DeepFace: Closing the gap to human-level performance in face verification. In CVPR'2014.
723. Tandy, D. W. (1997). Works and Days: A Translation and Commentary for the Social Sciences. University of California Press.
724. Tang, Y. and Eliasmith, C. (2010). Deep networks for robust visual recognition. In Proceedings of the 27th International Conference on Machine Learning, June 21–24, 2010, Haifa, Israel.
725. Tang, Y., Salakhutdinov, R., and Hinton, G. (2012). Deep mixtures of factor analysers. arXiv preprint arXiv:1206.4635.
726. Taylor, G. and Hinton, G. (2009). Factored conditional restricted Boltzmann machines for modeling motion style. In L. Bottou and M. Littman, editors, Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09), pages 1025–1032, Montreal, Quebec, Canada. ACM.
727. Taylor, G., Hinton, G. E., and Roweis, S. (2007). Modeling human motion using binary latent variables. In B. Schölkopf, J. Platt, and T. Hoffman, editors, Advances in Neural Information Processing Systems 19 (NIPS'06), pages 1345–1352. MIT Press, Cambridge, MA.
728. Teh, Y., Welling, M., Osindero, S., and Hinton, G. E. (2003). Energy-based models for sparse overcomplete representations. Journal of Machine Learning Research, 4, 1235–1260.
729. Tenenbaum, J., de Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. Science, 290(5500), 2319–2323.
730. Theis, L., van den Oord, A., and Bethge, M. (2015). A note on the evaluation of generative models. arXiv:1511.01844.
731. Thompson, J., Jain, A., LeCun, Y., and Bregler, C. (2014). Joint training of a convolutional network and a graphical model for human pose estimation. In NIPS'2014.
732. Thrun, S. (1995). Learning to play the game of chess. In NIPS'1994.
733. Tibshirani, R. J. (1995). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society B, 58, 267–288.
734. Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08), pages 1064–1071. ACM.
735. Tieleman, T. and Hinton, G. (2009). Using fast weights to improve persistent contrastive divergence. In L. Bottou and M. Littman, editors, Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09), pages 1033–1040. ACM.
736. Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal components analysis. Journal of the Royal Statistical Society B, 61(3), 611–622.
737. Torralba, A., Fergus, R., and Weiss, Y. (2008). Small codes and large databases for recognition. In Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'08), pages 1–8.

738. Touretzky, D. S. and Minton, G. E. (1985). Symbols among the neurons: Details of a connectionist inference architecture. In Proceedings of the 9th International Joint Conference on Artificial Intelligence – Volume 1, IJCAI'85, pages 238–243, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
739. Töscher, A., Jahrer, M., and Bell, R. M. (2009). The BigChaos solution to the Netflix grand prize.
740. Tu, K. and Honavar, V. (2011). On the utility of curricula in unsupervised learning of probabilistic grammars. In IJCAI'2011.
741. Turaga, S. C., Murray, J. F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., Denk, W., and Seung, H. S. (2010). Convolutional networks can learn to generate affinity graphs for image segmentation. Neural Computation, 22(2), 511–538.
742. Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: A simple and general method for semi-supervised learning. In Proc. ACL'2010, pages 384–394.
743. Uria, B., Murray, I., and Larochelle, H. (2013). Rnade: The real-valued neural autoregressive density-estimator. In NIPS'2013.
744. van den Oörd, A., Dieleman, S., and Schrauwen, B. (2013). Deep content-based music recommendation. In NIPS'2013.
745. van der Maaten, L. and Hinton, G. E. (2008). Visualizing data using t-SNE. J. Machine Learning Res., 9.
746. Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on CPUs. In Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop.
747. Vapnik, V. N. (1982). Estimation of Dependences Based on Empirical Data. Springer- Verlag, Berlin.
748. Vapnik, V. N. (1995). The Nature of Statistical Learning Theory. Springer, New York.
749. Vapnik, V. N. and Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability and Its Applications, 16, 264–280.
750. Vincent, P. (2011). A connection between score matching and denoising autoencoders. Neural Computation, 23(7).
751. Vincent, P. and Bengio, Y. (2003). Manifold Parzen windows. In NIPS'2002. MIT Press.
752. Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In ICML 2008.
753. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. J. Machine Learning Res., 11.
754. Vincent, P., de Brébisson, A., and Bouthillier, X. (2015). Efficient exact gradient update for training deep networks with very large sparse targets. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, Advances in Neural Information Processing Systems 28, pages 1108–1116. Curran Associates, Inc.
755. Vinyals, O., Kaiser, L., Koo, T., Petrov, S., Sutskever, I., and Hinton, G. (2014a). Grammar as a foreign language. Technical report, arXiv:1412.7449.
756. Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2014b). Show and tell: a neural image caption generator. arXiv 1411.4555.
757. Vinyals, O., Fortunato, M., and Jaitly, N. (2015a). Pointer networks. arXiv preprint arXiv:1506.03134.

758. Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015b). Show and tell: a neural image caption generator. In CVPR'2015. arXiv:1411.4555.
759. Viola, P. and Jones, M. (2001). Robust real-time object detection. In International Journal of Computer Vision.
760. Visin, F., Kastner, K., Cho, K., Matteucci, M., Courville, A., and Bengio, Y. (2015). ReNet: A recurrent neural network based alternative to convolutional networks. arXiv preprint arXiv:1505.00393.
761. Von Melchner, L., Pallas, S. L., and Sur, M. (2000). Visual behaviour mediated by retinal projections directed to the auditory pathway. Nature, 404(6780), 871–876.
762. Wager, S., Wang, S., and Liang, P. (2013). Dropout training as adaptive regularization. In Advances in Neural Information Processing Systems 26, pages 351–359.
763. Waibel, A., Hanazawa, T., Hinton, G. E., Shikano, K., and Lang, K. (1989). Phoneme recognition using time-delay neural networks. IEEE Transactions on Acoustics, Speech, and Signal Processing, 37, 328–339.
764. Wan, L., Zeiler, M., Zhang, S., LeCun, Y., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In ICML'2013.
765. Wang, S. and Manning, C. (2013). Fast dropout training. In ICML'2013.
766. Wang, Z., Zhang, J., Feng, J., and Chen, Z. (2014a). Knowledge graph and text jointly embedding. In Proc. EMNLP'2014.
767. Wang, Z., Zhang, J., Feng, J., and Chen, Z. (2014b). Knowledge graph embedding by translating on hyperplanes. In Proc. AAAI'2014.
768. Warde-Farley, D., Goodfellow, I. J., Courville, A., and Bengio, Y. (2014). An empirical analysis of dropout in piecewise linear networks. In ICLR'2014.
769. Wawrzynek, J., Asanovic, K., Kingsbury, B., Johnson, D., Beck, J., and Morgan, N. (1996). Spert-II: A vector microprocessor system. Computer, 29(3), 79–86.
770. Weaver, L. and Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. In Proc. UAI'2001, pages 538–545.
771. Weinberger, K. Q. and Saul, L. K. (2004). Unsupervised learning of image manifolds by semidefinite programming. In CVPR'2004, pages 988–995.
772. Weiss, Y., Torralba, A., and Fergus, R. (2008). Spectral hashing. In NIPS, pages 1753–1760.
773. Welling, M., Zemel, R. S., and Hinton, G. E. (2002). Self supervised boosting. In Advances in Neural Information Processing Systems, pages 665–672.
774. Welling, M., Hinton, G. E., and Osindero, S. (2003a). Learning sparse topographic representations with products of Student t-distributions. In NIPS'2002.
775. Welling, M., Zemel, R., and Hinton, G. E. (2003b). Self-supervised boosting. In S. Becker, S. Thrun, and K. Obermayer, editors, Advances in Neural Information Processing Systems 15 (NIPS'02), pages 665–672. MIT Press.
776. Welling, M., Rosen-Zvi, M., and Hinton, G. E. (2005). Exponential family harmoniums with an application to information retrieval. In L. Saul, Y. Weiss, and L. Bottou, editors, Advances in Neural Information Processing Systems 17 (NIPS'04), volume 17, Cambridge, MA. MIT Press.
777. Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In Proceedings of the 10th IFIP Conference, 31.8–4.9, NYC, pages 762–770.
778. Weston, J., Bengio, S., and Usunier, N. (2010). Large scale image annotation: learning to rank with joint word-image embeddings. Machine Learning, 81(1), 21–35.

779. Weston, J., Chopra, S., and Bordes, A. (2014). Memory networks. arXiv preprint arXiv:1410.3916.
780. Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In 1960 IRE WESCON Convention Record, volume 4, pages 96–104. IRE, New York.
781. Wikipedia (2015). List of animals by number of neurons – Wikipedia, the free encyclopedia. [Online; accessed 4-March-2015].
782. Williams, C. K. I. and Agakov, F. V. (2002). Products of Gaussians and Probabilistic Minor Component Analysis. Neural Computation, 14(5), 1169–1182.
783. Williams, C. K. I. and Rasmussen, C. E. (1996). Gaussian processes for regression. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, Advances in Neural Information Processing Systems 8 (NIPS'95), pages 514–520. MIT Press, Cambridge, MA.
784. Williams, R. J. (1992). Simple statistical gradient-following algorithms connectionist reinforcement learning. Machine Learning, 8, 229–256.
785. Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. Neural Computation, 1, 270–280.
786. Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. Neural Networks, 16(10), 1429–1451.
787. Wilson, J. R. (1984). Variance reduction techniques for digital simulation. American Journal of Mathematical and Management Sciences, 4(3), 277–312.
788. Wiskott, L. and Sejnowski, T. J. (2002). Slow feature analysis: Unsupervised learning of invariances. Neural Computation, 14(4), 715–770.
789. Wolpert, D. and MacReady, W. (1997). No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1, 67–82.
790. Wolpert, D. H. (1996). The lack of a priori distinction between learning algorithms. Neural Computation, 8(7), 1341–1390.
791. Wu, R., Yan, S., Shan, Y., Dang, Q., and Sun, G. (2015). Deep image: Scaling up image recognition. arXiv:1501.02876.
792. Wu, Z. (1997). Global continuation for distance geometry problems. SIAM Journal of Optimization, 7, 814–836.
793. Xiong, H. Y., Barash, Y., and Frey, B. J. (2011). Bayesian prediction of tissue-regulated splicing using RNA sequence and cellular context. Bioinformatics, 27(18), 2554–2562.
794. Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In ICML'2015, arXiv:1502.03044 .
795. Yildiz, I. B., Jaeger, H., and Kiebel, S. J. (2012). Re-visiting the echo state property. Neural networks, 35, 1–9.
796. Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In NIPS'2014.
797. Younes, L. (1998). On the convergence of Markovian stochastic algorithms with rapidly decreasing ergodicity rates. In Stochastics and Stochastics Models, pages 177–228.
798. Yu, D., Wang, S., and Deng, L. (2010). Sequential labeling using deep-structured conditional random fields. IEEE Journal of Selected Topics in Signal Processing.
799. Zaremba, W. and Sutskever, I. (2014). Learning to execute. arXiv 1410.4615.
800. Zaremba, W. and Sutskever, I. (2015). Reinforcement learning neural Turing machines. arXiv:1505.00521.

801. Zaslavsky, T. (1975). Facing Up to Arrangements: Face-Count Formulas for Partitions of Space by Hyperplanes. Number no. 154 in Memoirs of the American Mathematical Society. American Mathematical Society.
802. Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In ECCV'14.
803. Zeiler, M. D., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., and Hinton, G. E. (2013). On rectified linear units for speech processing. In ICASSP 2013.
804. Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2015). Object detectors emerge in deep scene CNNs. ICLR'2015, arXiv:1412.6856.
805. Zhou, J. and Troyanskaya, O. G. (2014). Deep supervised and convolutional generative stochastic network for protein secondary structure prediction. In ICML'2014.
806. Zhou, Y. and Chellappa, R. (1988). Computation of optical flow using a neural network. In Neural Networks, 1988., IEEE International Conference on, pages 71–78. IEEE.
807. Zöhrer, M. and Pernkopf, F. (2014). General stochastic networks for classification. In NIPS'2014.

Предметный указатель

P

C

Ян Гудфеллоу, Иошуа Бенджио, Аарон Курвилль

Глубокое обучение

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 61,125. Тираж 200 экз.

Веб-сайт издательства: **www.дмк.рф**