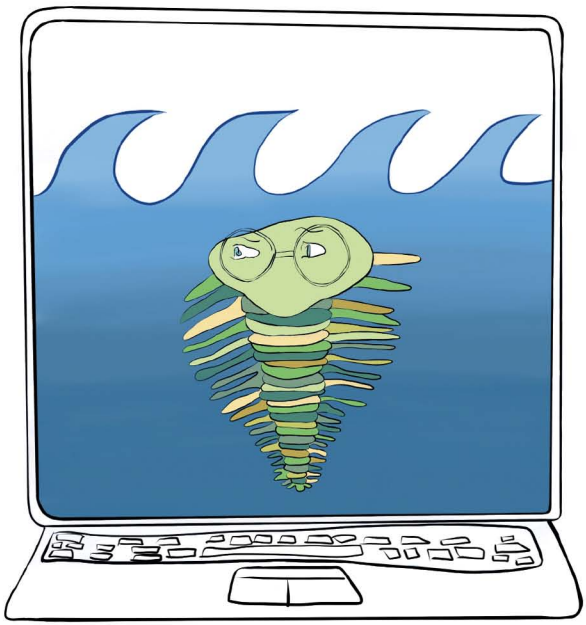
Addison Wesley Data & Analytics Series

ГЛУБОКОЕ ОБУЧЕНИЕ В КАРТИНКАХ

Визуальный гид по искусственному интеллекту



ДЖОН КРОН
ГРАНТ БЕЙЛЕВЕЛЬД и АГЛАЭ БАССЕНС

Deep Learning Illustrated

A Visual, Interactive Guide to Artificial Intelligence

Jon Krohn
with Grant Beyleveld
and Aglaé Bassens

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

ГЛУБОКОЕ ОБУЧЕНИЕ В КАРТИНКАХ

Визуальный гид по искусственному интеллекту

ДЖОН КРОН
ГРАНТ БЕЙЛЕВЕЛЬД и АГЛАЭ БАССЕНС

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.813
УДК 004.8
К83

Крон Джон, Бейлевельд Грант, Аглаэ Бассенс

К83 Глубокое обучение в картинках. Визуальный гид по искусственному интеллекту. — СПб.: Питер, 2020. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1574-7

Глубокое обучение стало мощным двигателем для работы с искусственным интеллектом. Яркие иллюстрации и простые примеры кода избавят вас от необходимости вникать в сложные аспекты конструирования моделей глубокого обучения, делая сложные задачи доступными и увлекательными.

Джон Крон, Грант Бейлевельд и замечательный иллюстратор Аглаэ Бассенс используют яркие примеры и аналогии, которые позволяют объяснить, что такое глубокое обучение, почему оно пользуется такой популярностью и как эта концепция связана с другими подходами к машинному обучению. Книга идеально подойдет разработчикам, специалистам по обработке данных, исследователям, аналитикам и начинающим программистам, которые хотят применять глубокое обучение в своей работе. Теоретические выкладки прекрасно дополняются прикладным кодом на Python в блокнотах Jupyter. Вы узнаете приемы создания эффективных моделей в TensorFlow и Keras, а также познакомитесь с PyTorch.

Базовые знания о глубоком обучении позволят создавать реальные приложения — от компьютерного зрения и обработки естественного языка до генерации изображений и игровых алгоритмов.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.813
УДК 004.8

ISBN 978-0135116692 англ.
ISBN 978-5-4461-1574-7

ОГЛАВЛЕНИЕ

ОТЗЫВЫ О КНИГЕ «ГЛУБОКОЕ ОБУЧЕНИЕ В КАРТИНКАХ»

«В течение ближайших десятилетий искусственный интеллект может кардинально изменить практически каждый аспект нашей жизни, во многом благодаря нынешним прорывам в глубоком обучении. Ясный и наглядный стиль изложения, выбранный авторами, позволяет получить исчерпывающее представление о современных возможностях искусственных нейронных сетей, а также о грядущем волшебстве».

Тим Урбан (Tim Urban), автор и иллюстратор сайта Wait But Why[1]

«Эта книга является доступным, практичным и обширным введением в глубокое обучение, а также одной из самых красиво иллюстрированных книг по машинному обучению».

Доктор Майкл Осборн (Michael Osborne), доцент дайсоновского технологического института кафедры машинного обучения, Оксфордский университет

«Эта книга — превосходное введение в глубокое обучение для начинающих, так как содержит множество конкретных и простых примеров с соответствующими учебными видеороликами и блокнотами Jupyter. Настоятельно рекомендуется всем интересующимся».

Доктор Чонг Ли (Chong Li), сооснователь компании Nakamoto & Turing Labs; адъюнкт-профессор Колумбийского университета

---

[1] Версия сайта на русском языке: https://wbwtranslations.com/. — *Примеч. пер.*

«Сегодня сложно представить разработку новых продуктов, создатели которых не задумывались бы о включении в них механизмов машинного обучения. Глубокое обучение имеет много практических применений, и эта книга благодаря ясному и наглядному стилю изложения пригодится всем, кто хотел бы понять, что такое глубокое обучение и как оно может повлиять на их работу и жизнь в будущем».

Хелен Альтшулер (Helen Altshuler), ведущий инженер Google

«Красивые иллюстрации и забавные аналогии в этой книге чрезвычайно упрощают освоение теории глубокого обучения. Простые примеры кода и практические советы позволяют читателям немедленно применить прогрессивные приемы к своей сфере интересов».

Доктор Расмус Ротэ (Rasmus Rothe), основатель компании Merantix

«Это бесценный источник знаний для всех, кто хочет понять, что такое глубокое обучение и почему сегодня оно используется практически во всех автоматизированных приложениях, от чат-ботов и инструментов распознавания голоса до автомобилей с автоматическим управлением. Иллюстрации и аналогии из области биологии помогают вдохнуть жизнь в эту сложную тему и облегчают понимание основных понятий».

Джошуа Марч (Joshua March), генеральный директор и сооснователь компании Conversocial; автор книги «Message Me»

«Глубокое обучение регулярно преображает положение дел в таких сферах, как компьютерное зрение, обработка естественного языка и задачи принятия последовательных решений. Если вы тоже хотите передавать данные через глубокие нейронные сети и получать высокопроизводительные модели, то эта книга с ее инновационным и наглядным подходом — идеальный старт».

Доктор Алекс Флинт (Alex Flint), робототехник и предприниматель

Моей Джиджи.
— Джон

ПРЕДИСЛОВИЕ

Многие считают машинное обучение будущим статистики и компьютерной инженерии, поскольку оно оказывает существенное влияние на обслуживание клиентов, проектирование, банковское дело, медицину, промышленное производство и множество других дисциплин и отраслей. Пока трудно переоценить его влияние на мир и те изменения, которые оно повлечет в ближайшие годы и десятилетия. Из множества методов машинного обучения, применяемых профессионалами (например, регрессия со штрафом, случайные леса и расширяемые деревья), наибольший энтузиазм вызывает глубокое обучение.

Глубокое обучение произвело революцию в компьютерном зрении и обработке естественного языка, а исследователи продолжают находить новые области, которые можно преобразить с помощью нейронных сетей. Наиболее сильное его влияние проявляется в попытках воспроизвести человеческий опыт, такой как зрение, восприятие языка, а также синтез речи и перевод текстов с одного языка на другой. Но математика и понятия, лежащие в основе глубокого обучения, могут показаться слишком пугающими, чтобы начать его использовать.

Авторы книги «Глубокое обучение в картинках» бросают вызов традиционным препятствиям и передают свои знания в простой и доступной форме, чтобы вам было приятно читать. Так же, как и другие книги этой серии («R for Everyone», «Pandas for Everyone», «Programming Skills for Data Science» и «Machine Learning with Python for Everyone»), эта книга отличается дружелюбием и доступностью для широкой аудитории. Математические формулы используются минимально, а там, где они приводятся, их сопровождает простое и понятное объяснение. Большинство идей дополнены наглядными иллюстрациями и программным кодом, использующим библиотеку *Keras*, который также доступен в виде простых для понимания блокнотов *Jupyter*.

Джон Крон вот уже много лет преподает глубокое обучение и известен своими яркими выступлениями на Нью-Йоркской конференции по статистическому программированию «New York Open Statistical Programming Meetup», где встречаются члены того же сообщества, на основе которого он создал свою группу по глубокому обучению «Deep Learning Study Group». Его владение предметом прослеживается в стиле изложения: он дает читателям достаточный

объем знаний и в то же время позволяет им получить удовольствие от чтения. К нему присоединились Грант Бейлевельд и Аглаэ Бассенс, которые вместе с Джоном делятся своим опытом применения алгоритмов глубокого обучения и подготовили для книги мастерски выполненные рисунки.

«Глубокое обучение в картинках» содержит исчерпывающее описание глубокого обучения, сочетая теорию, математические основы, где это необходимо, код и иллюстрации. Книга полностью охватывает рассматриваемую тему, включая полносвязанные сети, сверточные, рекуррентные, генеративно-состязательные нейронные сети и обучение с подкреплением, а также сферы их применения. Благодаря этому она станет идеальным помощником для тех, кто хочет познакомиться с нейронными сетями и получить практическое руководство по их реализации. Любой может (и должен) получать не только выгоду, но и удовольствие от времени, проведенного за чтением вместе с Джоном, Грантом и Аглаэ.

Джаред Ландер (Jared Lander),
редактор серии

ВСТУПЛЕНИЕ

Вашу нервную систему составляют миллиарды взаимосвязанных *нейронов*, которые также часто называют *клетками головного мозга*. Именно они позволяют вам чувствовать, думать и действовать. Методично окрашивая и исследуя тонкие срезы мозговой ткани, испанский врач Сантьяго Кахал (*Santiago Cajal*, рис. В.1) стал первым[1], кто идентифицировал нейроны (рис. В.2), и в первой половине двадцатого столетия исследователи начали проливать свет на то, как работают эти биологические клетки. К 1950-м годам ученые, вдохновленные новыми знаниями о работе мозга, экспериментировали с компьютерными *искусственными* нейронами, связывая их друг с другом и формируя *искусственные нейронные сети*, которые в общих чертах имитируют работу естественных тезок.

Рис. В.1. Сантьяго Кахал (1852–1934)

Опираясь на эту краткую историю, мы можем дать обманчиво простое определение термину «глубокое обучение»: глубокое обучение предполагает создание сети, в которой искусственные нейроны — как правило, тысячи, миллионы и даже больше — распределены, по крайней мере, по нескольким слоям. Первый слой искусственных нейронов передает информацию второму слою, второй — третьему, и так далее, пока последний слой не выведет некоторые значения. Однако, как будет показано в этой книге, это простое определение недостаточно удовлетворительно отражает потрясающую широту возможностей глубокого обучения и некоторые его нюансы.

---

[1] *Кахал С.* (*Cajal, S.-R.*). «Les Nouvelles Idées sur la Structure du Système Nerveux chez l'Homme et chez les Vertébrés» Paris (1894), C. Reinwald & Companie.

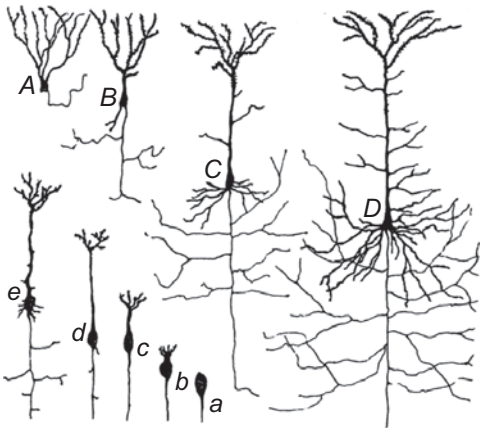


Рис. В.2. Диаграмма из работы Кахала (1894), иллюстрирующая рост нейрона (*a*–*e*) и сравнение нейронов лягушки (*A*), ящерицы (*B*), крысы (*C*) и человека (*D*)

В главе 1 мы подробно расскажем, как с появлением достаточно недорогой вычислительной техники, способной обрабатывать большие наборы данных, и нескольких важных теоретических достижений, первая волна цунами глубокого обучения, достигшая берега, показала выдающиеся результаты в состязании по компьютерному зрению в 2012 году. Ученые и специалисты заметили это, и в последующие годы, насыщенные событиями, глубокое обучение проникло в бесчисленное множество стандартных приложений. От автопилота в автомобилях *Tesla* до распознавания голоса в помощнике *Amazon Alexa*, от перевода текстов в реальном времени с одного языка на другой до интеграции с сотнями продуктов *Google*, глубокое обучение позволило повысить точность множества вычислительных задач с 95 до 99% и даже выше — эти несколько процентов могут создать ощущение, что автоматизированный сервис действует как по волшебству. Интерактивные примеры кода, представленные в этой книге, развеют это кажущееся волшебство, и тем не менее глубокое обучение действительно наполнило машины сверхчеловеческими способностями, позволяющими им решать такие сложные задачи, как распознавание лиц, обобщение текста и поиск выигрышных стратегий в сложных настольных играх[1]. Учитывая эти выдающиеся достижения, неудивительно, что термин «глубокое обучение» стал синонимом «искусственного интеллекта» в популярной прессе, на работе и дома.

Это было потрясающее время, потому что, как вы узнаете в этой книге, только раз в жизни бывает, чтобы распространенное понятие вызвало такой интерес за такой короткий промежуток времени. Мы рады, что вы тоже проявили интерес к глубокому обучению, и не можем дождаться, чтобы поделиться нашим энтузиазмом относительно этой беспрецедентно прогрессивной сферы.

[1] Сравнительный обзор возможностей машин и человека вы найдете по адресу http://bit.ly/aiindex18.

КАК ПОЛЬЗОВАТЬСЯ ЭТОЙ КНИГОЙ

Книга разделена на четыре части. Часть I «Введение в глубокое обучение» ориентирована на всех заинтересованных читателей. В ней приводится общий обзор глубокого обучения, его развития и связей с такими понятиями, как искусственный интеллект (ИИ), машинное обучение и обучение с подкреплением. Этот раздел насыщен яркими иллюстрациями, простыми аналогиями и короткими историями. Он будет интересен всем, даже не имеющим опыта программирования.

Части II–IV, напротив, ориентированы на разработчиков программного обеспечения, специалистов по обработке данных, исследователей, аналитиков и всех тех, кто хотел бы научиться применять методы глубокого обучения в своей области. Эти части предлагают теоретический фундамент с минимумом математических формул и опорой на простые, наглядные аналогии и практические примеры на Python. Кроме того, возможность опробовать действующий код, который доступен в сопутствующих блокнотах Jupyter1, способствует практическому пониманию основных подходов к применению глубокого обучения: компьютерное зрение (глава 10), обработка естественного языка (глава 11), генерирование изображений (глава 12) и игры (глава 13). Во избежание разночтений программный код в книге будет оформлен `моноширинным шрифтом, например так`. Также для большего удобства мы использовали стили оформления, принятые в *Jupyter* по умолчанию.

Тем, кому описание математических и статистических основ глубокого обучения в этой книге покажется недостаточно глубоким, мы рекомендуем два наших любимых источника для дальнейшего изучения:

1. Электронную книгу Майкла Нильсена (*Michael Nielsen*) «*Neural Networks and Deep Learning*»,[2] в которой используются небольшие и забавные интерактивные апплеты для демонстрации понятий и математическая нотация, подобная нашей.
2. Книгу Яна Гудфеллоу (*Ian Goodfellow*) (мы познакомимся с ним в главе 3), Иошуа Бенджио (*Yoshua Bengio*) (см. рис. 1.10) и Аарона Курвилля (*Aaron Courville*) «*Deep Learning*»[3], в которой достаточно полно рассматриваются математические основы нейронных сетей.

---

[1] github.com/the-deep-learners/deep-learning-illustrated

[2] *Nielsen M.* (2015). «Neural Networks and Deep Learning», Determination Press. Доступна бесплатно по адресу *neuralnetworksanddeeplearning.com*

[3] *Goodfellow I.* et al. (2016). «Deep Learning». MIT Press. Доступна бесплатно по адресу *deeplearningbook.org*. (*Гудфеллоу Ян, Бенджио Иошуа, Курвилль Аарон*. Глубокое обучение. ДМК Пресс, 2017, ISBN: 978-5-97060-554-7. — *Примеч. пер.*)

На протяжении всей книги вам будут встречаться добродушные трилобиты, предлагающие прочитать дополнительные сведения, которые, по их мнению, могут быть интересны или полезны. *Читающий трилобит* (как на рис. Р.3) — это книжный червь, предлагающий расширить ваши познания. *Трилобит, привлекающий внимание* (как на рис. Р.4), постарается внести ясность в те фрагменты текста, которые могут вызвать сомнения. Кроме трилобитов, оформленных как врезки, мы также использовали сноски. Их не обязательно читать, но они содержат краткие объяснения новых терминов и сокращений, а также цитаты из оригинальных статей и источники, к которым вы сможете обратиться, если пожелаете.

Рис. Р.3. Читающий трилобит любит расширять ваши познания

Рис. Р.4. Этот трилобит пытается привлечь внимание к сложным объяснениям в тексте. Не игнорируйте его!

Многое из того, что описывается в книге, изложено также в соответствующих видеоуроках. Формат книги позволил нам подробно изложить теоретические основы, но, просмотрев видеоуроки, вы сможете взглянуть на наши блокноты *Jupyter* с другой стороны, когда каждая строка кода объясняется вслух по мере ввода[1]. Видеоуроки разбиты на три серии, каждая из которых соответствует главам книги:

1. «*Deep Learning with TensorFlow LiveLessons*»[2]: главы 1 и 5–10.
2. «*Deep Learning for Natural Language Processing LiveLessons*»[3]: главы 2 и 11.
3. «*Deep Reinforcement Learning and GANs LiveLessons*»[4]: главы 3, 4, 12 и 13.

---

[1] Многие блокноты *Jupyter*, представленные в этой книге, взяты непосредственно из видеоуроков и были написаны до книги. Кое-где мы обновили код, готовя его для публикации, поэтому иногда книжные версии кода и версии в видеоуроках могут не совпадать в деталях, хотя в целом будут действовать одинаково.

[2] *Krohn J.* (2017). «Deep Learning with TensorFlow LiveLessons: Applications of Deep Neural Networks to Machine Learning Tasks» (video course). Boston: Addison-Wesley.

[3] *Krohn J.* (2017). «Deep Learning for Natural Language Processing LiveLessons: Applications of Deep Neural Networks to Machine Learning Tasks» (video course). Boston: Addison-Wesley.

[4] *Krohn J.* (2018). «Deep Reinforcement Learning and GANs LiveLessons: Advanced Topics in Deep Learning» (video course). Boston: Addison-Wesley.

БЛАГОДАРНОСТИ

Мы благодарны команде *Untapt*, особенно Эндрю Влахутину (Andrew Vlahutin), Сэму Кенни (Sam Kenny) и Винсу Петаччо II (*Vince Petaccio II*), которые поддерживали нас, пока мы писали эту книгу. Отдельное спасибо влюбленному в нейронные сети Эду Доннеру (Ed Donner), который поддерживал нашу страсть к глубокому обучению.

Мы также благодарны всем членам группы по глубокому обучению «*Deep Learning Study Group*»[1], регулярно посещающим наши живые и энергичные занятия в офисах *Untapt* в Нью-Йорке. Эта книга рождалась в ходе обсуждений в этой группе и сейчас уже трудно представить, что она могла зародиться как-то иначе.

Спасибо нашим научным редакторам за бесценные отзывы, способствовавшие заметному улучшению книги: Алексу Липатову (*Alex Lipatov*), Эндрю Влахутину (*Andrew Vlahutin*), Клаудии Перлич (*Claudia Perlich*), Дмитрию Нестеренко (*Dmitri Nesterenko*), Джейсону Байку (*Jason Baik*), Лауре Грессер (*Laura Graesser*), Майклу Гриффитсу (*Michael Griffiths*), Полу Диксу (*Paul Dix*) и Ва Луну Кенгу (*Wah Loon Keng*). Спасибо редакторам и руководителям издательства: Крису Зану (*Chris Zahn*), Бетси Хардингер (*Betsy Hardinger*), Энн Попик (*Anna Popick*) и Джулии Нахиль (*Julie Nahil*), чьи дотошность и продуманность обеспечили высокое качество, ясность и представительность книги. Спасибо Джареду Ландеру (*Jared Lander*), который возглавляет открытое сообщество статистического программирования в Нью-Йорке, присоединился к нашей группе по глубокому обучению и помог встретиться с Деброй Уильямс Коули (*Debra Williams Cauley*) из издательства *Pearson*. Выражаем особую благодарность самой Дебре, поддержавшей нашу идею публикации в первую же встречу и сыгравшей важную роль в успехе книги. Также спасибо ученым и специалистам по машинному обучению, которые учили нас теоретическим основам и продолжают вдохновлять, особенно Джонатану Флинту (*Jonathan Flint*), Феликсу Агакову (*Felix Agakov*) и Уиллу Валдару (*Will Valdar*).

Наконец, выражаем огромную благодарность нашим семьям и друзьям, которые не только смирились с тем, что мы продолжали работать по выходным и в отпуске, но и всячески поддерживали нас.

[1] deeplearningstudygroup.org

ОБ АВТОРАХ

Джон Крон (Jon Krohn) — главный специалист по обработке данных в компании Untapt, занимающейся вопросами машинного обучения. Ведет популярные серии видеоуроков, опубликованные издательством *Addison-Wesley*, в том числе «*Deep Learning* с *TensorFlow LiveLessons*» и «*Deep Learning for Natural Language Processing LiveLessons*». Ведет курс о глубоком обучении в Нью-Йоркской академии наук о данных и читает лекции в Колумбийском университете. Имеет докторскую степень по неврологии, полученную в Оксфордском университете, и с 2010 года публикует материалы по машинному обучению в ведущих рецензируемых журналах, включая «*Advances in Neural Information Processing Systems*».

Грант Бейлевельд (Grant Beyleveld), специалист по обработке данных, в настоящее время занимается темой обработки естественного языка с использованием глубокого изучения. Получил докторскую степень в области биомедицинских наук в Медицинской школе Икан при клинике Маунт-Синай в Нью-Йорке, исследовав отношения между вирусами и их носителями. Является одним из основателей *deeplearningstudygroup.org*.

Аглаэ Бассенс (Aglaé Bassens) — художница, родившаяся в Бельгии и теперь живущая в Париже. Училась в школе искусств Раскина при Оксфордском университете и в Лондонском университете изящных искусств. Помимо создания иллюстраций она также занимается живописью, натюрмортами и фресками.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

I

ВВЕДЕНИЕ В ГЛУБОКОЕ ОБУЧЕНИЕ

1

БИОЛОГИЧЕСКОЕ И КОМПЬЮТЕРНОЕ ЗРЕНИЕ

На протяжении всей этой главы и большей части книги мы будем рассказывать о глубоком обучении, опираясь на систему зрения биологических организмов как аналогию, чтобы оживить наше повествование. Кроме общего представления о глубоком обучении, эта аналогия поможет понять, почему приемы глубокого обучения обладают такой мощью и настолько широко применимы.

БИОЛОГИЧЕСКОЕ ЗРЕНИЕ

Пятьсот пятьдесят миллионов лет тому назад, в доисторический кембрийский период, число видов живых организмов на планете резко увеличилось (рис. 1.1). Изучение окаменелостей дало понять[1], что этот всплеск был вызван появлением светочувствительных клеток у трилобитов, небольших морских организмов, родственников современных крабов (рис. 1.2). Система зрения, пусть даже примитивная, дарует восхитительный набор новых возможностей. Можно, например, заметить еду, врагов и дружелюбных товарищей на некотором расстоянии. Другие органы чувств, например обоняния, тоже позволяют обнаруживать все это, но не так точно и быстро, как органы зрения. Согласно гипотезе, появление зрения у трилобитов привело к гонке за выживание, способствовавшей кембрийскому взрыву: жертвы трилобитов, а также хищники, питавшиеся ими, вынуждены были эволюционировать, чтобы выжить.

За полмиллиарда лет, прошедших с той поры, когда трилобиты развили зрение, сложность этих органов чувств значительно возросла. Действительно,

[1] *Parker A.* (2004). «In the Blink of an Eye: How Vision Sparked the Big Bang of Evolution». New York: Basic Books.

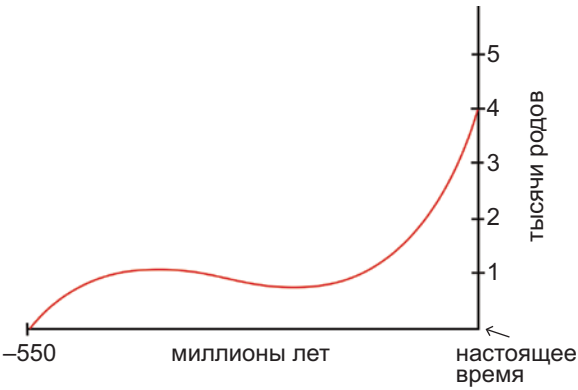


Рис. 1.1. 550 миллионов лет тому назад, в кембрийский период, число видов животных на нашей планете стало резко увеличиваться. «Роды» — это категории родственных видов

Рис. 1.2. Очковый трилобит

у современных млекопитающих в зрительном восприятии[1] принимает участие значительная часть *коры головного мозга* — внешнее серое вещество. В конце 1950-х годов в университете Джона Хопкинса физиологи Дэвид Хьюбел (*David Hubel*) и Торстен Визель (*Torsten Wiesel*, рис. 1.3) начали проводить свои новаторские исследования, чтобы выяснить, как обрабатывается визуальная информация в коре головного мозга млекопитающих[2] (за это они впоследствии были удостоены Нобелевской премии[3]). Как показано на рис. 1.4, Хьюбел и Визель

[1] Вот пара интересных фактов о коре головного мозга: во-первых, это одно из последних эволюционных изменений мозга, способствовавшее усложнению поведения млекопитающих, по сравнению с поведением более ранних классов животных, таких как рептилии и амфибии. Во-вторых, обычно, говоря о мозге, подразумевают *серое вещество*, потому что кора головного мозга — тонкая внешняя его оболочка — имеет серый цвет, однако основную часть мозга составляет *белое вещество*. Белое вещество отвечает за перенос информации на большие расстояния, потому что составляющие его нейроны покрыты жироподобным веществом белого цвета, ускоряющим передачу сигналов. В качестве грубой аналогии нейроны в белом веществе можно рассматривать как «магистрали». Эти высокоскоростные автомагистрали почти не имеют съездов, зато могут очень быстро передавать сигнал из одной части мозга в другую. Напротив, «местные дороги» серого вещества образуют разветвленную сеть, открывая широчайшие возможности для взаимодействий между нейронами, хотя и с меньшей скоростью. То есть если говорить обобщенно, кора головного мозга — серое вещество — это та его часть, где происходят самые сложные вычисления, обеспечивающие сложное поведение крупных животных, таких как млекопитающие, в частности, больших приматов вроде *Homo sapiens*.

[2] *Hubel D. H. & Wiesel T. N.* (1959). «Receptive fields of single neurones in the cat's striate cortex». The Journal of Physiology, 148, 574–91.

[3] Нобелевская премия 1981 года по физиологии и медицине, поделенная с американским нейробиологом Роджером Сперри (Roger Sperry).

проводили исследования, демонстрируя изображения кошкам, находившимся под воздействием анестезии, и одновременно регистрируя активность отдельных нейронов *первичной зрительной коры*, передней части коры головного мозга, получающей нервные импульсы от глаз.

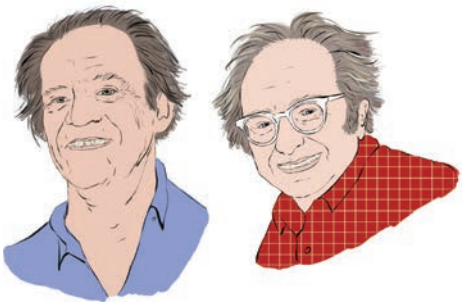


Рис. 1.3. Лауреаты Нобелевской премии нейрофизиологи: Торстен Визель, слева и Дэвид Хьюбел, справа

Хьюбел и Визель начали с простых фигур, таких как точка (рис. 1.4), проецируя их на экран перед кошками. Первые результаты разочаровывали: нейроны первичной зрительной коры никак не реагировали на изображения. Они были обескуражены тем, что клетки, которые анатомически являются воротами для передачи визуальной информации в остальную часть коры головного мозга, не реагируют на зрительные раздражители. В отчаянии Хьюбел и Визель тщетно пытались стимулировать нейроны, прыгая и размахивая руками перед кошкой. Ничего. Но затем, как это не раз бывало с великими открытиями, от рентгеновских лучей до пенициллина в микроволновой печи, Хьюбел и Визель

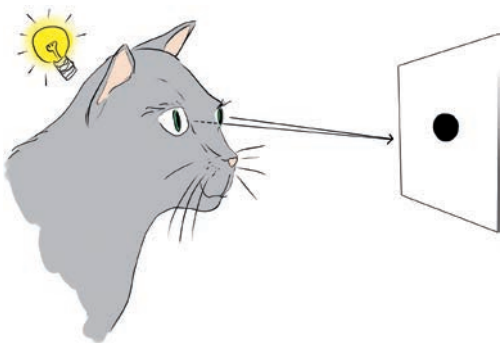


Рис. 1.4. Хьюбел и Визель демонстрировали изображения кошкам, используя слайды и проектор, и одновременно регистрировали активность нейронов в первичной зрительной коре мозга. В своих экспериментах они имплантировали электрическое записывающее оборудование непосредственно в череп кошки. Мы не стали показывать, как это выглядело, посчитав, что нагляднее будет представить его как лампочку. На этом рисунке показано, что по счастливой случайности нейрон первичной зрительной коры головного мозга активировался, когда срез слайда пересекал освещенную область

совершенно случайно кое-что заметили: при пересечении срезом слайда освещенной области, когда они вынимали его из проектора, послышалось отчетливое потрескивание записывающего оборудования, предупреждающее, что активизировался нейрон первичной зрительной коры головного мозга. От радости они принялись носиться как сумасшедшие по лабораторным коридорам университета Джона Хопкинса.

Случайный треск, вызванный активностью нейрона, не был аномалией. В ходе дальнейших экспериментов Хьюбел и Визель обнаружили, что нейроны, которые получают сигнал от глаза, в целом наиболее чувствительны к простым прямым краям. Поэтому они назвали их *простыми* нейронами.

Как показано на рис. 1.5, Хьюбел и Визель определили, что этот простой нейрон оптимально реагирует на край с определенной ориентацией. Большая группа простых нейронов, каждый из которых специализируется на определенной ориентации, может распознавать края с любой ориентацией. Эти простые клетки передают информацию большому количеству так называемых *сложных* нейронов. Сложный нейрон получает визуальную информацию, которая уже была обработана несколькими простыми клетками, поэтому он готов к объединению линий с разной ориентацией в более сложные формы, такие как угол или кривая.

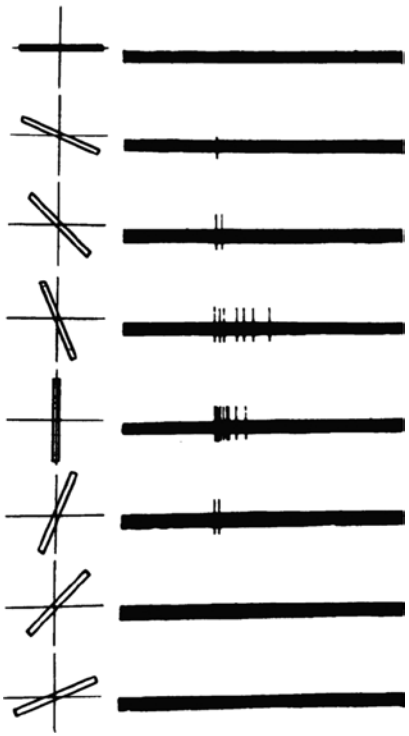


Рис. 1.5. Простая клетка в первичной зрительной коре кошки срабатывает с разной частотой, в зависимости от ориентации видимой линии. Линии с разной ориентацией показаны в левом столбце, а в правом столбце показана электрическая активность клетки с течением времени (одна секунда). Вертикальная линия (в пятом ряду сверху) вызывает наибольшую электрическую активность в этой конкретной простой клетке. Линии, немного отклоняющиеся от вертикали (в промежуточных рядах), вызывают меньшую активность, а линии с ориентацией, близкой к горизонтали (в верхнем и нижнем рядах), — незначительную или нулевую активность

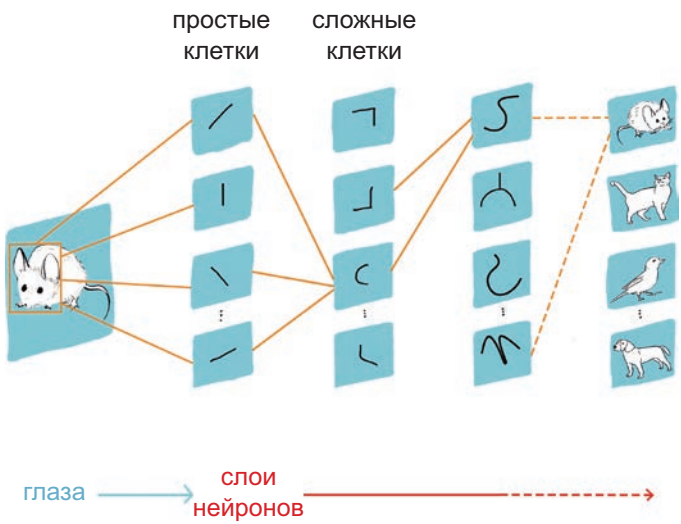


Рис. 1.6. Иллюстрация того, как последовательные слои биологических нейронов представляют визуальную информацию в мозге, например, кошки или человека

На рис. 1.6 показано, как благодаря множеству иерархически организованных слоев нейронов, передающих информацию нейронам более высокого порядка, мозг может представлять все более сложные зрительные образы. В поле зрения попадает изображение головы мыши. Фотоны света стимулируют нейроны, находящиеся в сетчатке каждого глаза, и эта необработанная визуальная информация передается от глаз к первичной зрительной коре головного мозга. Там первый слой нейронов, *простые клетки* Хьюбела и Визеля, получающие эту информацию, обнаруживают края (прямые линии) в определенных ориентациях. Таких нейронов многие тысячи, но в целях упрощения на рис. 1.6 показаны только четыре. Эти простые нейроны передают информацию о наличии или отсутствии линий в определенных ориентациях следующему слою *сложных клеток*, которые принимают и рекомбинируют информацию, формируя более сложные зрительные абстракции, такие как округлая форма головы мыши.

По мере передачи информации через последующие слои зрительные абстракции постепенно становятся более сложными. Как иллюстрирует крайний правый слой нейронов, после такой иерархической обработки информации множеством слоев (мы использовали пунктирную стрелку, чтобы показать, что многие слои не показаны на рисунке) мозг оказывается способен представлять визуальные понятия в виде абстрактных понятий, таких как «мышь», «кошка», «птица» или «собака».

В настоящее время благодаря бесчисленным накопленным записям активности корковых нейронов пациентов, перенесших операции на головном мозге, а также данным, полученным неинвазивными методами, такими как магнитно-резо-

нансная томография (МРТ)[1], нейробиологи составили достаточно подробную карту областей, специализирующихся на обработке определенных зрительных абстракций, например цвет, движение и лица (рис. 1.7).

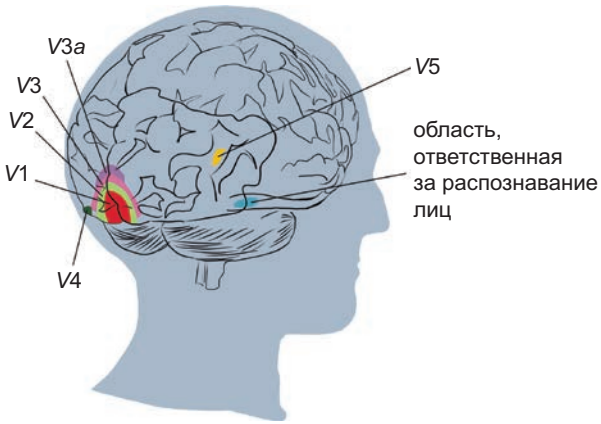


Рис. 1.7. Области зрительной коры. Область *V1* получает данные от глаз и содержит простые клетки, которые определяют ориентацию краев. Посредством рекомбинации информации множеством последующих слоев нейронов (в том числе в областях *V2*, *V3* и *V3a*) мозг формирует все более абстрактные визуальные образы. В человеческом мозге (показанном здесь) есть области, содержащие специализированные нейроны, например, для распознавания цвета (*V4*), движения (*V5*) и лиц людей (область, ответственная за распознавание лиц)

КОМПЬЮТЕРНОЕ ЗРЕНИЕ

Мы рассмотрели биологическую систему зрения не только потому, что это интересно (надеюсь, что предыдущий раздел действительно был вам интересен), но и потому, что она послужила прообразом для современных подходов глубокого обучения к компьютерному зрению.

На рис. 1.8 представлена краткая хронология развития зрения у биологических организмов и методов компьютерного зрения. На верхней шкале, выделенной синим цветом, отмечены моменты появления зрения у трилобитов и публикации статьи Хьюбела и Визеля в 1959 году об иерархической природе первичной зрительной коры головного мозга, о чем рассказывалось в предыдущем разделе. Шкала, представляющая развитие компьютерного зрения, разделена на два параллельных потока, соответствующие двум альтернативным подходам. Средняя шкала розового цвета представляет подход на основе глубокого обу-

[1] Особенно *функциональная* МРТ, которая позволяет выяснить, какие участки коры головного мозга особенно активны или неактивны, когда мозг занимается определенной деятельностью.

чения, описываемого в этой книге. Нижняя шкала фиолетового цвета представляет традиционный путь развития компьютерного зрения на основе обычного машинного обучения, сопоставление с которым поможет лучше понять мощь и революционный характер глубокого обучения.

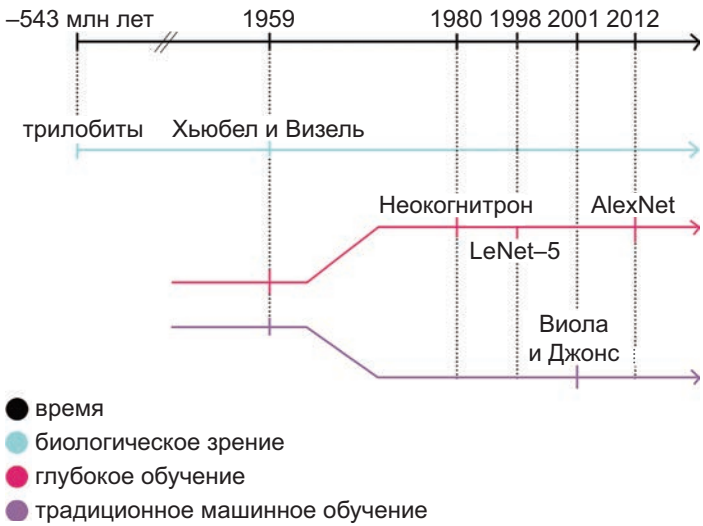


Рис. 1.8. Краткая хронология биологического и компьютерного зрения с ключевыми историческими событиями в подходах к компьютерному зрению с позиций глубокого обучения и традиционного машинного обучения, которые рассматриваются в этом разделе

НЕОКОГНИТРОН

В конце 1970-х японский инженер-электрик Кунихико Фукусима (Kunihiko Fukushima), вдохновленный открытием простых и сложных клеток, образующих иерархию первичной зрительной коры, предложил аналогичную архитектуру для реализации компьютерного зрения, которую назвал *неокогнитроном*[1]. Отметим два важных момента:

1. В своих работах Фукусима прямо упомянул работу Хьюбела и Визеля. Фактически он ссылается на три их статьи по организации первичной зрительной коры и заимствует определение «простых» и «сложных» клеток для описания первого и второго слоев своего неокогнитрона соответственно.

[1] *Fukushima K.* (1980). «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». Biological Cynbernetics, 36, 193–202.

2. При организации в иерархическую структуру искусственные нейроны[1], ближе расположенные к входному сигналу — как и их биологические аналоги на рис. 1.6, — распознают линии с разной ориентацией, а находящиеся в более глубоких слоях распознают все более сложные абстрактные объекты. Чтобы продемонстрировать это мощное свойство неокогнитрона и его потомков, реализованных с использованием приемов глубокого обучения, мы рассмотрим интерактивный пример в конце этой главы[2].

LENET-5

Неокогнитрон был способен, например, идентифицировать рукописные символы[3], но затем появилась модель *LeNet-5*[4], созданная Яном Лекуном (Yann LeCun, рис. 1.9) и Йошуа Бенжио (Yoshua Bengio, рис. 1.10), точность и эффективность которой сделали ее значимым событием. Иерархическая архитектура LeNet-5 (рис. 1.11) основана на работе Фукусимы и открытии Хьюбела

Рис. 1.9. Уроженец Парижа Ян Лекун — одна из выдающихся фигур в сфере искусственных нейронных сетей и исследований глубокого обучения. Лекун является директором и основателем Нью-Йоркского университетского центра наук о данных, а также руководит исследованиями в области ИИ в социальной сети Facebook

Рис. 1.10. Йошуа Бенжио — еще один из ведущих исследователей в сфере искусственных нейронных сетей и глубокого обучения. Родился во Франции. Профессор информатики в Монреальском университете, он руководит известной программой «Машины и разум» в Канадском институте перспективных исследований

[1] Точное определение термину «искусственные нейроны» будет дано в главе 7. А пока просто считайте каждый искусственный нейрон маленьким быстрым алгоритмом.

[2] В частности, на рис. 1.19 показана эта иерархия с последовательно усложняющимися представлениями.

[3] *Fukushima K. & Wake N.* (1991). «Handwritten alphanumeric character recognition by the neocognitron». IEEE Transactions on Neural Networks, 2, 355–65.

[4] *LeCun Y.* et al. (1998). «Gradient-based learning applied to document recognition». Proceedings of the IEEE, 2, 355–65.

и Визеля[1]. Кроме того, Лекун и его коллеги обладали превосходным набором данных для обучения своей модели[2], мощной вычислительной техникой и, что очень важно, алгоритмом обратного распространения.

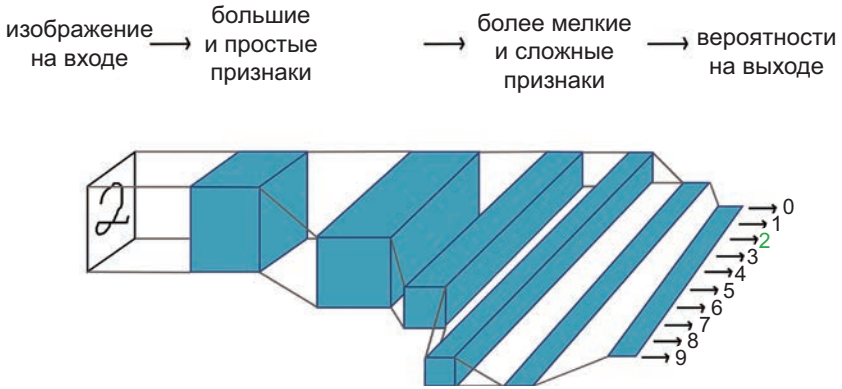


Рис. 1.11. Модель LeNet-5 имеет ту же иерархическую архитектуру, что была обнаружена в первичной зрительной коре Хьюбелом и Визелем и использована Фукусимой в его неокогнитроне. В ней точно так же самый левый слой распознает простые границы, а последующие слои формируют все более сложные элементы. При таком способе обработки информации рукописная «2» правильно распознается как цифра два (выделена зеленым в выходах справа)

Обратное распространение способствует эффективному обучению искусственных нейронов во всех слоях в модели глубокого обучения[3]. Наряду с правильным набором данных и мощной вычислительной техникой алгоритм обратного распространения сделал модель LeNet-5 достаточно надежной, чтобы впервые применить глубокое обучение в коммерческих целях: она использовалась почтовой службой США для автоматизации чтения почтовых индексов на конвертах. В главе 10, посвященной компьютерному зрению, мы поближе познакомимся с моделью LeNet-5, разработав ее и научив распознавать рукописные цифры.

В LeNet-5 Ян Лекун с коллегами реализовали алгоритм, способный правильно предсказывать рукописные цифры и не требовавший включать какие-либо знания о рукописных цифрах в код. Как следствие, LeNet-5 дала возможность провести фундаментальное различие между глубоким и традиционным машинным обучением. Как показано на рис. 1.12, в традиционном подходе к машинному обучению основные усилия затрачиваются на *проектирование признаков*. Обычно для этого к исходным данным применяются сложные алгоритмы для

---

[1] LeNet-5 стала первой *сверточной нейронной сетью* — разновидностью глубокого обучения, доминирующей в современном компьютерном зрении, которая подробно рассматривается в главе 10.

[2] Их набор данных, ставший классическим (рукописные цифры MNIST), широко используется в части II «Теоретические основы в картинках».

[3] Алгоритм обратного распространения мы исследуем в главе 7.

их предварительной обработки и преобразования во входные переменные, которые легко моделируются традиционными статистическими методами. Эти методы, такие как регрессия, случайный лес и метод опорных векторов, редко бывают эффективными при применении к исходным данным, и поэтому проектирование входных данных исторически было одной из основных целей для профессионалов машинного обучения.

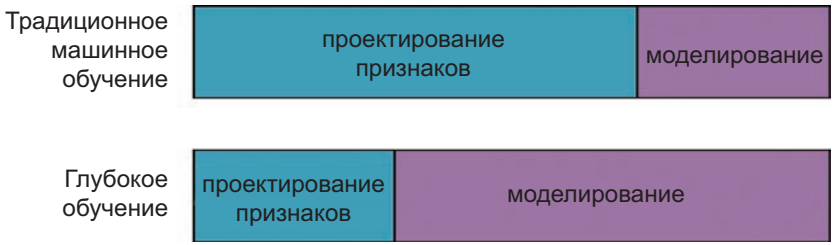


Рис. 1.12. Проектирование признаков — преобразование исходных данных во входные переменные — часто преобладает в сфере традиционного машинного обучения. В глубоком обучении этап проектирования признаков почти полностью отсутствует, а большая часть времени расходуется на разработку и настройку моделей

В общем случае на оптимизацию и выбор наиболее эффективных моделей специалист, практикующий традиционное машинное обучение, тратит меньшую часть времени. В глубоком обучении приоритеты меняются местами. *Специалист по глубокому обучению почти не занимается проектированием признаков и почти все свое время использует для моделирования данных с помощью разных искусственных нейронных сетей, которые автоматически преобразуют исходные данные в полезные признаки*. Различие между глубоким и традиционным машинным обучением является основной темой этой книги. В следующем разделе представлен классический пример проектирования признаков, чтобы прояснить различие.

ТРАДИЦИОННОЕ МАШИННОЕ ОБУЧЕНИЕ

После появления LeNet-5 исследования в области искусственных нейронных сетей, включая глубокое обучение, замерли. Многие решили, что автоматическое формирование признаков в этом подходе непрактично, даже при том что оно показало впечатляющие результаты в распознавании рукописных символов. Идея отсутствия признаков воспринималась как ограничение широты применения[1]. Традиционное машинное обучение, включающее проектирование

[1] В то время имелись некоторые проблемы, связанные с оптимизацией моделей глубокого обучения, которые впоследствии были решены, включая проблему инициализации весов плохими значениями (описанную в главе 9), сдвиг переменных (также описывается в главе 9) и преобладание относительно неэффективной сигмоидной функции активации (глава 6).

признаков, казалось более перспективным направлением, и финансирование глубокого обучения сократилось[1].

Чтобы было понятнее, что такое проектирование признаков, на рис. 1.13 показан знаменитый пример Пола Виолы (Paul Viola) и Майкла Джонса (Michael Jones), представленный ими в начале 2000-х[2]. Виола и Джонс использовали прямоугольные фильтры — вертикальные или горизонтальные черно-белые полосы, как показано на рис. 1.13. Признаки, получаемые при перемещении этих фильтров по изображению, включались в алгоритмы машинного обучения для надежного распознавания лица. Эта работа примечательна тем, что алгоритм достаточно эффективно распознавал лица в масштабе реального времени в сферах за пределами биологии[3].

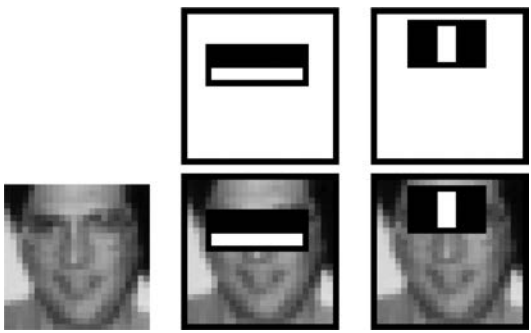


Рис. 1.13. Подход к проектированию признаков, предпринятый Виолой и Джонсом (2001), обеспечивающий надежное распознавание лиц. Их эффективный алгоритм нашел применение в камерах Fujifilm, облегчив автоматическую фокусировку в масштабе реального времени

Хорошие фильтры распознавания лиц для преобразования исходных пикселов в признаки и последующей их передачи в модель машинного обучения были получены в результате многолетних исследований и совместной работы. Конечно же, этот алгоритм распознает лица только как таковые — он не способен отличить, например, Ангелу Меркель от Опры Уинфри. Чтобы спроектировать признаки, которые позволят отличить лицо Опры в частности, или некоторые объекты, не являющиеся лицами, такие как дома, автомобили или йоркширские терьеры, необходимо накопить опыт в решении этой категории задач, что может снова потребовать годы работы академического сообщества. Эх, если бы мы могли как-то обойтись без таких трудозатрат!

---

[1] Государственное финансирование исследований в области искусственных нейронных сетей сократилось во всем мире, за исключением канадского федерального правительства, которое продолжало оказывать заметную поддержку, что позволило университетам Монреаля, Торонто и Альберты занять лидирующие позиции в этой области.

[2] *Viola P. & Jones M.* (2001). «Robust real-time face detection». International Journal of Computer Vision, 57, 137–54.

[3] Спустя несколько лет алгоритм нашел применение в цифровых камерах Fujifilm, упростив автоматическую фокусировку на лицах. Ныне это привычный атрибут цифровых камер и смартфонов.

IMAGENET И ILSVRC

Как отмечалось выше, одним из преимуществ LeNet-5 перед неокогнитроном был более объемный набор высококачественных обучающих данных. Следующему прорыву в сфере нейронных сетей также способствовал общедоступный набор высококачественных данных, но на этот раз значительно большего размера. *ImageNet* — это маркированный каталог фотографий, разработанный Фей-Фей Ли (Fei-Fei Li, рис. 1.14), предоставляющий исследователям компьютерного зрения огромный каталог обучающих данных[1,2]. Для справки: набор с рукописными цифрами, использованный для обучения LeNet-5, содержал несколько десятков тысяч изображений. ImageNet содержит десятки *миллионов*.

Рис. 1.14. Потрясающий набор данных ImageNet был создан китайско-американским профессором информатики Фей-Фей Ли и ее коллегами из Принстона в 2009 году. В настоящее время она преподает в Стэнфордском университете и одновременно руководит исследованиями в области искусственного интеллекта и машинного обучения для облачной платформы Google

14 миллионов изображений в наборе ImageNet распределены по 22 000 категорий, например: контейнеровозы, леопарды, морские звезды, растения. В 2010 году Ли организовала открытый конкурс под названием ILSVRC (ImageNet Large Scale Visual Recognition Challenge — состязание по масштабному распознаванию образов на основе ImageNet) на подмножестве данных ImageNet, ставший главной площадкой для оценки самых современных алгоритмов компьютерного зрения. Подмножество ILSVRC включает 1.4 миллиона изображений из 1000 категорий. В довесок к широкому диапазону многие из предложенных категорий содержат изображения разных пород собак, что позволяет оценить способность алгоритмов различать не только сильно отличающиеся изображения, но и довольно похожие между собой[3].

---

[1] image-net.org

[2] *Deng J.* et al. (2009). «ImageNet: A large-scale hierarchical image database». Proceedings of the Conference on Computer Vision and Pattern Recognition.

[3] Попробуйте как-нибудь на досуге отличить по фотографиям йоркширского терьера от австралийского шелковистого терьера. Это сложно, но судьи Вестминстерской выставки собак, а также современные модели компьютерного зрения способны на это. Между прочим, большое количество фотографий с собаками в этом наборе стало причиной склонности моделей глубокого обучения, обученных на наборе ImageNet, «мечтать» о собаках (см., например, deepdreamgenerator.com).

ALEXNET

Как показано на рис. 1.15, в первые два года существования конкурса ILSVRC все алгоритмы, участвовавшие в состязании, основывались на идеологии традиционного машинного обучения с проектированием признаков. На третий год все участники, *кроме одного*, вновь использовали традиционные алгоритмы машинного обучения. Если бы эта единственная модель глубокого обучения не была разработана в 2012 году или ее создатели не приняли участия в ILSVRC, точность классификации изображений по сравнению с предыдущим годом увеличилась бы незначительно. Однако Алекс Крижевский (Alex Krizhevsky) и Илья Суцкевер (Ilya Sutskever) из лаборатории Университета Торонто под руководством Джеффри Хинтона (Geoffrey Hinton, рис. 1.16) разгромили соперников с помощью своей разработки, которую ныне называют AlexNet (рис. 1.17)[1,2]. Это событие стало решающим. В одно мгновение на передний план

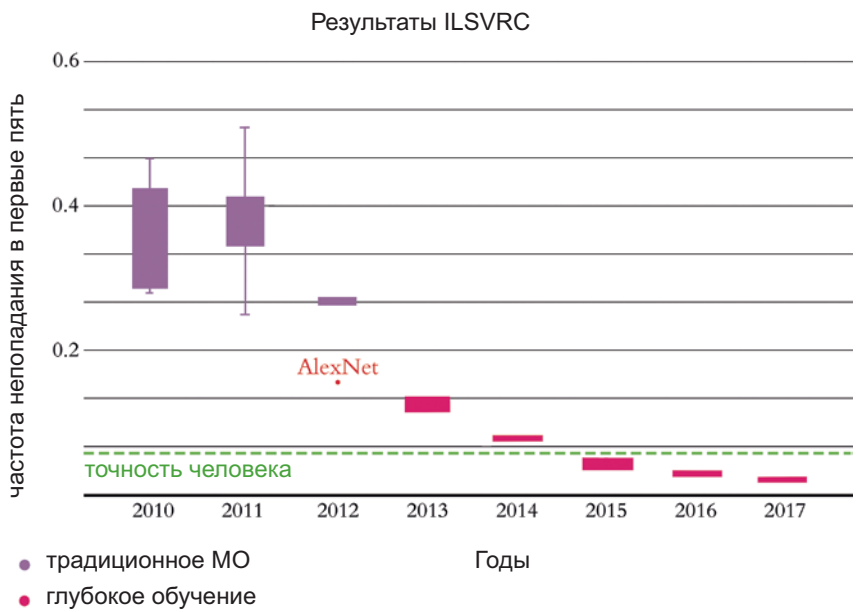


Рис. 1.15. Лучшие достижения в конкурсе ILSVRC по годам. Модель AlexNet стала абсолютным победителем в состязании 2012 года (с отрывом в 40%!). С тех пор наивысшую точность показывали только алгоритмы глубокого обучения. В 2015 году компьютеры превзошли человека в точности распознавания

[1] *Krizhevsky A., Sutskever I. & Hinton G.* (2012). «ImageNet classification with deep convolutional neural networks». Advances in Neural Information Processing Systems, 25.

[2] Изображения в нижней части рис. 1.17 взяты из статьи Иосински Дж. (Yosinski J.) и др. (2015). «Understanding neural networks through deep visualization». *arXiv: 1506.06579*.

вышли архитектуры глубокого обучения. Ученые и специалисты, воодушевленные таким успехом, начали изучать основы искусственных нейронных сетей и создавать программные библиотеки — многие из них с открытым исходным кодом — для экспериментов с моделями глубокого обучения на собственных данных и для своих нужд, будь то компьютерное зрение или что-то другое. Как показано на рис. 1.15, начиная с 2012 года все самые эффективные модели, участвовавшие в конкурсе ILSVRC, были основаны на глубоком обучении.

Рис. 1.16. У истоков искусственных нейронных сетей стоял выдающийся британо-канадский ученый Джеффри Хинтон, которого в масс-медиа часто называют «крестным отцом глубокого обучения». Хинтон — почетный профессор Торонтского университета и научный руководитель исследовательской группы Brain Team поискового гиганта Google в Торонто. В 2019 году Хинтон, Ян Лекун (рис. 1.9) и Йошуа Бенжио (рис. 1.10) были удостоены премии Тьюринга — высшей награды в области информатики — за исследования в сфере глубокого обучения

Даже при том что своей архитектурой AlexNet напоминает LeNet-5, есть три важных фактора, позволивших ей занять лидирующие позиции среди алгоритмов компьютерного зрения в 2012 году. Первый — обучающие данные. Крижевский и его коллеги не только использовали фотографии из огромного каталога ImageNet, но и искусственно расширили доступные данные, применяя преобразования к обучающим образцам (мы этим тоже займемся в главе 10). Второй — вычислительная мощность. Мало того что с 1998 по 2012 год резко выросла мощность вычислительной техники на единицу стоимости, Крижевский, Хинтон и Суцкевер дополнительно использовали два графических процессора (GPU[1]), обеспечивших невиданную ранее эффективность обучения на больших наборах данных. Третий — архитектурные достижения. AlexNet имеет больше слоев, чем LeNet-5, и использует преимущества нового типа искусственных нейронов[2] и интересный трюк[3], который помогает модели глубокого обучения достичь обобщения данных, на которых она обучалась. В главе 10 мы с вами создадим свою модель AlexNet и используем ее для классификации изображений.

---

[1] Graphical Processing Unit — графический процессор; предназначен, главным образом, для отображения графики в видеоиграх, но также хорошо подходит для выполнения операций с матрицами, которые широко распространены в глубоком обучении в сотнях параллельных потоков.

[2] Блок линейной ректификации (Rectified Linear Unit, ReLU) будет представлен в главе 6.

[3] Прореживание (dropout) будет представлено в главе 9.

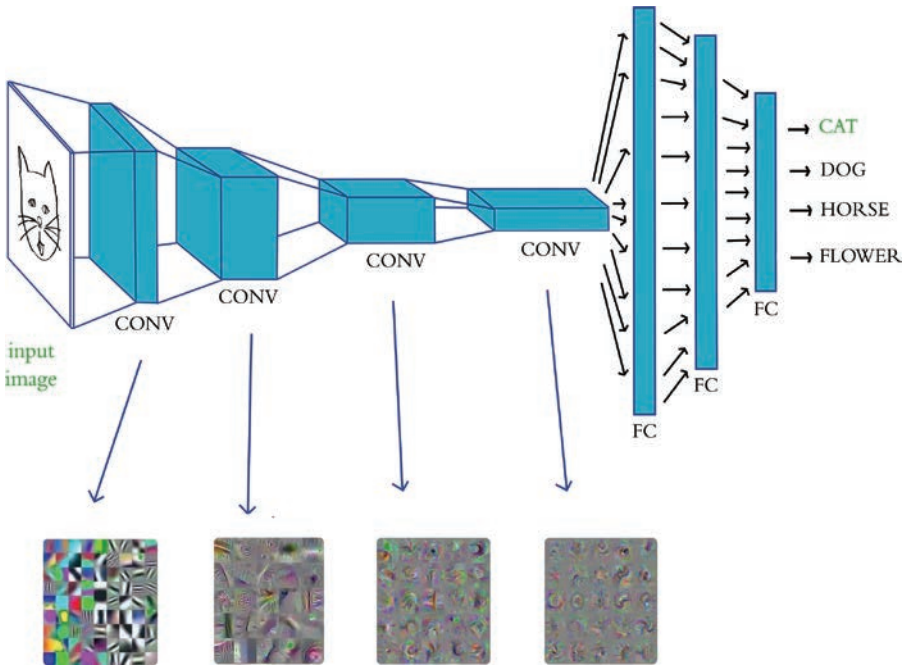


Рис. 1.17. Иерархическая архитектура AlexNet напоминает LeNet-5: первый слой (слева) распознает простые элементы изображений, такие как границы, а более глубокие — все более сложные и абстрактные понятия. Внизу показаны примеры изображений, на которые нейроны в соответствующих слоях реагируют с максимальной интенсивностью, напоминая тем самым слои биологической системы зрения на рис. 1.6 и демонстрируя иерархическое увеличение сложности распознаваемых образов. В показанном здесь примере изображение кошки, поданное на вход в LeNet-5, правильно идентифицировано как таковое (о чем свидетельствует зеленое слово «CAT» на выходе). Словом «CONV» отмечены сверточные (convolutional) слои, а словом «FC» — полносвязанные (fully connected) слои; с этими типами слоев мы познакомимся в главах 7 и 10 соответственно

Наш обзор ILSVRC подчеркивает сокрушительное преимущество моделей глубокого обучения, таких как AlexNet: благодаря им практически отпала необходимость иметь опыт в предметной области, чтобы создавать высокоточные прогностические модели. Эта тенденция перехода от проектирования признаков на основе опыта к удивительно мощным моделям глубокого обучения, генерирующим признаки автоматически, получила широкое распространение не только в компьютерном зрении, но и, например, в поиске выигрышных стратегий в сложных играх (тема главы 4), а также в обработке естественного языка (глава 2)[1]. Больше не нужно быть специалистом по визуальным особенностям лиц, чтобы создать алгоритм для их распознавания.

[1] Особенно интересный рассказ о прорывах в области машинного перевода можно найти в статье Гидеона Льюиса-Крауса (Gideon Lewis-Kraus) «The Great A. I. Awakening», опубликованной в журнале New York Times Magazine 14 декабря 2016.

Больше не нужно глубоко понимать стратегию игры, чтобы написать программу, которая способна справиться с ней. Больше не нужно быть авторитетным специалистом в структуре и семантике каждого из нескольких языков для разработки инструмента автоматического перевода с одного на другой. Для быстро растущего списка вариантов практического применения владение методами глубокого обучения перевешивает ценность знаний в предметной области. Если раньше такое умение могло потребовать докторской степени и даже опыта нескольких лет научных исследований, то на функциональном уровне владение глубоким обучением развить довольно легко — как при работе с этой книгой!

ИНТЕРАКТИВНАЯ СРЕДА TENSORFLOW

Чтобы в увлекательной и интерактивной форме познакомиться с иерархической природой глубокого обучения, которая может автоматически конструировать признаки, посетите интерактивную среду TensorFlow на *bit.ly/TFplayground*. Перейдя по этой специальной ссылке, вы получите сеть, как показано на рис. 1.18. Во второй части книги мы подробно опишем термины, которые отображаются на экране, а пока их можно просто проигнорировать. Прямо сейчас достаточно знать, что это — модель глубокого обучения. Она состоит из шести слоев искусственных нейронов: входной слой слева под заголовком FEATURES (Признаки), четыре слоя HIDDEN LAYERS (Скрытые слои), которые отвечают за обучение, и выходной слой OUTPUT (Выход), координатная сетка справа со значениями по осям координат от –6 до +6. Цель нейронной сети — научиться отличать оранжевые точки (отрицательные случаи) от синих (положительные случаи), опираясь исключительно на их местоположение в координатной сетке. То есть во входной слой для каждой точки передаются всего две координаты: горизонтальная (X_1) и вертикальная (X_2). Точки, используемые в роли обучающих данных, по умолчанию отображаются на координатной сетке. Установив флажок Show test data (Показать тестовые данные), можно увидеть точки, которые будут использоваться для оценки качества работы сети по мере обучения. Крайне важно, чтобы тестовые данные не были доступны во время обучения — они помогут нам получить сеть, которая хорошо обобщает новые, прежде не встречавшиеся данные.

Щелкните на кнопке Play/Pause (Запустить/Приостановить) в верхнем левом углу. Запустите обучение сети и продолжайте, пока оба значения в правом верхнем углу, Training loss (Величина потери на обучающих данных) и Test loss (Величина потери на тестовых данных), не приблизятся к нулю, скажем, опустятся ниже 0.05. Сколько времени это займет, зависит от мощности вашего компьютера, но, надеюсь, не более нескольких минут.

Как показано на рис. 1.19, искусственные нейроны, расположенные глубже (правее), распознают во входных данных все более сложные абстракции, так же как нейроны в неокогнитроне, LeNet-5 (см. рис. 1.11) и AlexNet (см. рис. 1.17).

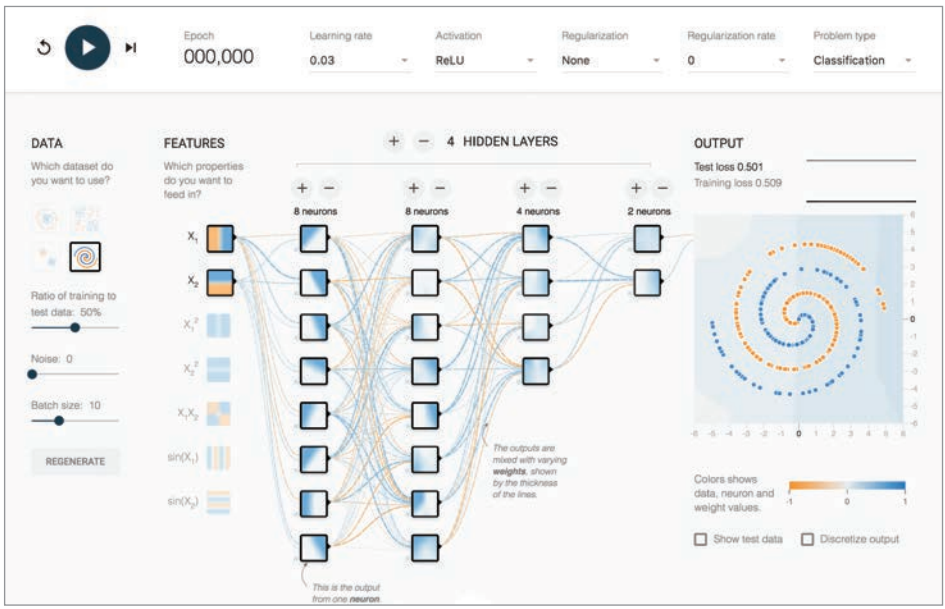


Рис. 1.18. Эта глубокая нейронная сеть учится отличать оранжевые точки (отрицательные случаи) от синих (положительные случаи), расположенных по спирали, опираясь только на их координаты по осям X_1 и X_2, как показано на координатной сетке справа

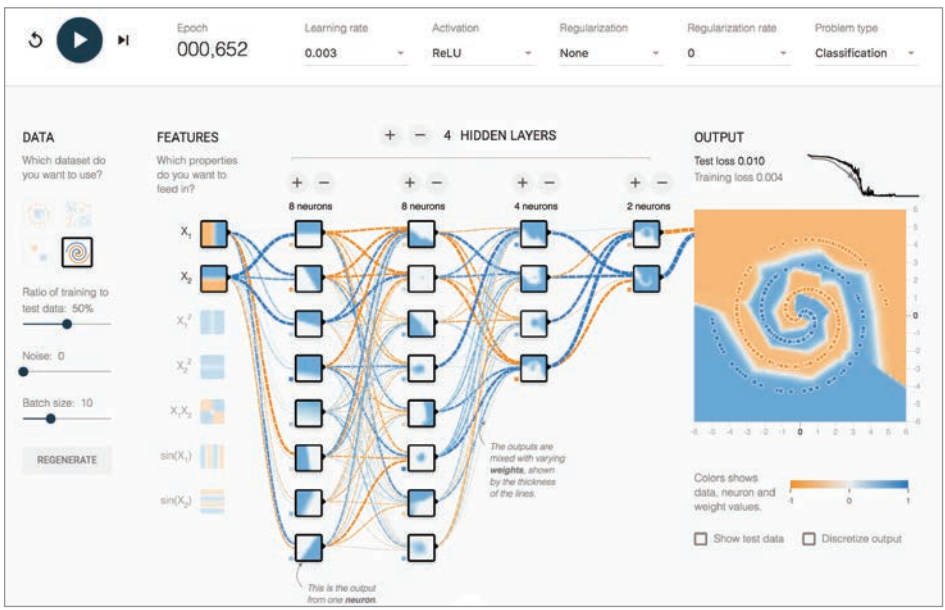


Рис. 1.19. Сеть после обучения

Каждый раз, когда запускается обучение, сеть приходит к решению задачи спиральной классификации уникальным путем, но сам подход остается неизменным (чтобы убедиться в этом, обновите страницу и повторите обучение сети). Искусственные нейроны в крайнем левом скрытом слое специализируются на распознавании краев (прямых линий), причем каждый из них имеет определенную ориентацию. Нейроны в первом скрытом слое передают информацию во второй, и каждый из них объединяет края в чуть более сложные абстракции, такие как кривые. Нейроны в каждом последующем слое рекомбинируют информацию, получаемую от предыдущих, постепенно увеличивая сложность и абстрактность представляемых признаков. В последнем слое (крайнем справа) нейроны способны представлять детали спиральной формы, что позволяет сети точно предсказать, является ли точка оранжевой (отрицательный случай) или синей (положительный случай) по ее координатам X_1 и X_2. Наведите указатель мыши на нейрон, чтобы спроецировать его на координатную сетку OUTPUT (Выход) и изучить его специализацию.

QUICK, DRAW!

Для интерактивного знакомства с сетью глубокого обучения, решающей задачу компьютерного зрения в масштабе реального времени, перейдите по ссылке quickdraw.withgoogle.com и сыграйте в игру *Quick, Draw!*. Щелкните на кнопке Let's Draw! (Начать), чтобы начать игру. Вам будет предложено нарисовать объект, а алгоритм глубокого обучения постарается угадать, что вы рисуете. К концу главы 10 мы рассмотрим все теоретические предпосылки и практические примеры кода, необходимые для разработки алгоритма компьютерного зрения, родственного этому. Кроме того, ваши наброски будут добавлены в набор данных, который мы используем в главе 12 для создания модели глубокого обучения, способной довольно точно имитировать человеческие рисунки. Держитесь крепче! Мы отправляемся в фантастическую поездку.

ИТОГИ

В этой главе мы познакомились с историей глубокого обучения от биологического прообраза до триумфального появления модели AlexNet в 2012 году, которая вывела методику глубокого обучения на сцену. Мы не раз повторили, что иерархическая архитектура моделей глубокого обучения позволяет им представлять все более сложные абстракции. Чтобы конкретизировать эту идею, мы завершили главу демонстрацией интерактивной среды TensorFlow, позволяющей обучить искусственную нейронную сеть и увидеть, как действуют иерархические представления. В следующей главе мы расширим эти идеи и перейдем от визуальных приложений к языковым.

2

ЯЗЫКИ ЛЮДЕЙ И МАШИН

В главе 1 мы в общих чертах познакомились с теорией глубокого обучения, проведя аналогию с системой биологического зрения. Одновременно мы подчеркнули, что одной из самых сильных сторон этого метода является его способность автоматически выявлять признаки на основе данных. Сейчас мы продолжим исследовать основы глубокого обучения на примере обработки естественного языка и особое внимание уделим способности автоматически изучать признаки, связанные со значениями слов.

Известный австро-британский философ Людвиг Витгенштейн в изданном посмертно труде «Философские исследования» утверждал: «Значение слова определяется его использованием в широком контексте языка»[1]. Он также писал: «Нельзя угадать, как функционирует слово. Нужно смотреть и запоминать, как оно используется». Витгенштейн предположил, что слова сами по себе не имеют реального значения; значение можно выяснить только по примерам их использования в более широком контексте. Как будет показано далее в этой главе, обработка естественного языка с применением методов глубокого обучения в значительной степени опирается на этот тезис. В действительности метод word2vec, который применяется для преобразования слов во входные данные числовой модели, явно получает семантические представления слов, анализируя их в контексте языка.

Вооружившись этой идеей, мы начнем с того, что выделим методику применения глубокого обучения к обработке естественного языка (Natural Language Processing, NLP) в отдельную дисциплину, а затем продолжим обсуждать современные методы, используемые для представления слов и языка. К концу

[1] *Wittgenstein L.* (1953). «Philosophical Investigations» (Anscombe G. Trans). Oxford, UK: Basil Blackwell. (*Людвиг Витгенштейн*. Философские исследования. АСТ Neoclassic, 2019. — *Примеч. пер.*)

этой главы мы получим достаточно полное понимание возможностей глубокого обучения и NLP и заложим основы для разработки моделей в главе 11.

ГЛУБОКОЕ ОБУЧЕНИЕ ДЛЯ ОБРАБОТКИ ЕСТЕСТВЕННОГО ЯЗЫКА

В этой главе мы рассмотрим два основных понятия: *глубокое обучение* и *обработка естественного языка*. Сначала рассмотрим нужные нам аспекты этих двух понятий по отдельности, а затем, по ходу изложения материала, сплетем их вместе.

СЕТИ ГЛУБОКОГО ОБУЧЕНИЯ АВТОМАТИЧЕСКИ ИЗУЧАЮТ ВАРИАНТЫ ПРЕДСТАВЛЕНИЯ

Как мы уже отмечали во вступлении, глубокое обучение можно определить как объединение в слои простых алгоритмов, называемых *искусственными нейронами*, и конструирование из них многослойных сетей. На диаграмме Венна (рис. 2.1) показано место глубокого обучения в семействе методов *обучения представлению*. Современное глубокое обучение занимает в этом семействе доминирующее положение, но оно также включает любые другие методы, основанные на автоматическом изучении особенностей данных. В действительности термины «признак» и «представление» взаимозаменяемы.

Схема на рис. 1.12 закладывает основу для понимания преимуществ обучения представлению перед традиционными подходами к машинному обучению. Традиционное машинное обучение, как правило, показывает хорошие результаты только благодаря умному коду, разработанному человеком: он преобразует исходные данные — изображения, аудиозаписи или текст — в признаки для передачи алгоритмам машинного обучения (таким как регрессия, случайный лес или метод опорных векторов), которые хорошо разбираются в методах взвешивания признаков, но не способны изучать признаки непосредственно по исходным данным. Такое формирование признаков вручную часто является узкоспециализированной задачей. Например, для работы с языковыми данными могут потребоваться знания лингвистики на уровне выпускников вузов.

Основное преимущество глубокого обучения — отсутствие требований к знаниям и опыту в предметной области. Здесь не требуется вручную извлекать входные признаки из исходных данных — в модель глубокого обучения можно передать непосредственно исходные данные. Анализируя большое число образцов, переданных в модель глубокого обучения, искусственные нейроны в первом слое сети на этих данных учатся представлять самые простые абстракции, а нейроны в каждом последующем слое обучаются представлять более сложные

нелинейные абстракции на тех, что были получены на предшествующем шаге. Как вы узнаете в этой главе, это не просто вопрос удобства; автоматическое изучение признаков дает дополнительные преимущества. Признаки, спроектированные человеком, как правило, не являются исчерпывающими, часто чрезмерно специфичны и требуют длительных циклов проектирования и проверки свойств, которые могут растянуться на годы. Модели обучения представлениям, напротив, генерируют признаки быстро (обычно в течение нескольких часов или дней обучения модели), легко адаптируются к изменениям в данных (например, к появлению новых слов, их значений и способов использования) и автоматически находят оптимальное решение задачи.

Рис. 2.1. Диаграмма Венна, представляющая место разных подходов в семействе методов машинного обучения

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

Обработка естественного языка (NLP) — это область исследований, расположенная на стыке информатики, лингвистики и искусственного интеллекта (рис. 2.2). NLP предполагает анализ образцов естественного разговорного или письменного языка (например, как предложение, которое вы читаете сейчас), и их обработку на компьютерах для решения какой-то задачи или ее упрощения. К числу примеров использования языка, не подпадающего под понятие *естественного*, можно отнести код, написанный на языке программирования, или короткие строки символов в электронной таблице.

Вот несколько примеров практического использования NLP:

- *Классификация документов*: использование языка документа (например, электронного письма, твита или отзыва к фильму) для определения его принадлежности к определенной категории (например, высокая срочность, позитивные настроения или прогнозируемое направление изменения цен на акции компании).
- *Машинный перевод*: помощь фирмам, занимающимся переводом с одного языка на другой, предложением машинного варианта перевода с исходного языка (например, английского) на целевой язык (например, немецкий или мандаринский язык); перевод, хотя и не всегда совершенный, может выполняться абсолютно автоматически.
- *Поисковые системы*: автоматическое дополнение поисковых запросов и прогнозирование характеристик искомой информации или веб-сайта.

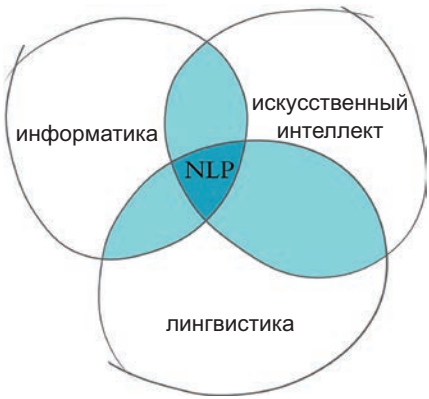


Рис. 2.2. NLP располагается на стыке информатики, лингвистики и искусственного интеллекта

- *Распознавание речи*: интерпретация голосовых команд для предоставления информации или принятия мер, как, например, в голосовых помощниках Amazon Alexa, Apple Siri или Microsoft Cortana.
- *Чат-боты*: поддержание естественной беседы в течение длительного времени; несмотря на недостаточное правдоподобие, чат-боты вполне успешно используются для поддержания относительно линейных диалогов на узкие темы, например, для разрешения типовых проблем при обращении в службу поддержки.

К наиболее простым задачам NLP относятся: проверка орфографии, подсказка синонимов и поиск по ключевым словам. Эти простые задачи довольно легко решаются с помощью детерминированного кода, основанного на правилах, например, из справочных словарей или предметных указателей. Модели глу-

бокого обучения неоправданно сложны для таких случаев, и поэтому мы их не обсуждаем в этой книге.

К задачам NLP относятся: определение уровня сложности документа, прогнозирование наиболее вероятных следующих слов при вводе запроса в поисковой системе, классификация документов (см. список выше) и извлечение из них такой информации, как цены или именованные сущности[1]. Эти варианты NLP средней сложности хорошо подходят для применения моделей глубокого обучения. Например, в главе 11 мы попробуем использовать различные архитектуры глубокого обучения для предсказания эмоциональной окраски отзывов к фильмам.

К наиболее сложным задачам NLP относятся: машинный перевод (см. список выше), автоматический выбор ответов на вопросы и чат-боты. Сложность в этих сферах обусловлена необходимостью учитывать массу тонкостей (например, юмор). Ответ на вопрос может зависеть от промежуточных ответов на предыдущие вопросы, и смысл вопроса может передаваться длинным отрывком текста, состоящим из множества предложений. Решение сложных задач NLP, подобных этим, выходит за рамки этой книги; однако основы, которые мы заложим, помогут вам продолжить развитие в этих направлениях.

КРАТКАЯ ИСТОРИЯ ГЛУБОКОГО ОБУЧЕНИЯ ДЛЯ NLP

Хронологическая шкала на рис. 2.3 отмечает последние вехи в применении глубокого обучения к задачам NLP. Начало было положено в 2011 году, когда ученый-компьютерщик из Университета Торонто Джордж Даль (George Dahl) и его коллеги из Microsoft Research совершили первый большой прорыв в области применения алгоритмов глубокого обучения к большим наборам данных[2]. Этот прорыв был совершен с использованием данных на естественном языке. Даль и его команда обучили глубокую нейронную сеть распознавать большое количество слов по аудиозаписям человеческой речи. Год спустя, как уже говорилось в главе 1, в Торонто произошло следующее знаменательное событие в области глубокого обучения: модель глубокого обучения AlexNet разгромила своих конкурентов в традиционном состязании по распознаванию изображений из масштабного каталога ImageNet (рис. 1.15). На тот момент эта модель компьютерного зрения, показавшая ошеломляющие результаты, наглядно проиллюстрировала преимущества глубокого обучения в задачах компьютерного зрения.

[1] К именованным сущностям относятся географические названия, имена известных лиц, названия компаний и продуктов.

[2] *Dahl G.* et al. (2011). «Large vocabulary continuous speech recognition with context-dependent DBN-HMMs» Proceedings of the International Conference on Acoustics, Speech, and Signal Processing.

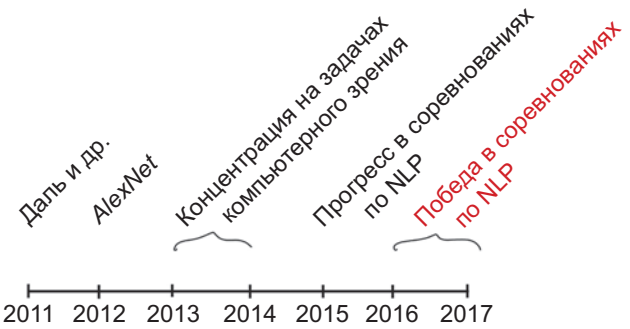


Рис. 2.3. Основные вехи на пути применения глубокого обучения к задачам обработки естественного языка

К 2015 году значительный прогресс, достигнутый в создании моделей компьютерного зрения, начал распространяться на соревнования по NLP — например, оценивающие точность машинных переводов с одного языка на другой. Модели глубокого обучения сумели приблизиться к точности традиционных методов машинного обучения; при этом они требовали меньше времени на исследования и разработку и имели меньшую вычислительную сложность. Фактически снижение вычислительной сложности позволило корпорации Microsoft создать программное обеспечение, выполняющее машинный перевод в масштабе реального времени на процессорах мобильных телефонов — потрясающий прогресс в решении задачи, для которой раньше требовалось подключение к Интернету и выполнение дорогостоящих вычислений на удаленном сервере. В 2016 и 2017 годах модели глубокого обучения, представленные на соревнованиях по NLP, не только оказались более эффективными, чем традиционные модели машинного обучения, но и превзошли их в точности. Посмотрим, как это произошло.

ВЫЧИСЛИТЕЛЬНОЕ ПРЕДСТАВЛЕНИЕ ЯЗЫКА

Для успешной обработки естественного языка его нужно передать в модель глубокого обучения в таком виде, чтобы та смогла его усвоить. Для любых компьютерных систем это означает количественное представление языка, например, в виде двумерной матрицы с числовыми значениями. В настоящее время особой популярностью пользуются два метода преобразования текста в числа — прямое кодирование (one-hot encoding) и векторы слов[1]. Сейчас мы обсудим оба метода по очереди.

[1] Если бы эта книга была посвящена задачам NLP, имело бы смысл упомянуть также методы, основанные на определении частоты слов, например TF-IDF (Term Frequency-Inverse Document Frequency — «частота слова — обратная частота документа») и PMI (Pointwise Mutual Information — «поточечная взаимная информация»).

ПРЯМОЕ КОДИРОВАНИЕ СЛОВ

Традиционно для числового кодирования естественного языка с целью его обработки на компьютере используется метод *прямого кодирования* (рис. 2.4). Согласно этому методу, слова естественного языка в предложении (например, «the», «bat», «sat», «on», «the» и «cat») представлены столбцами матрицы. При этом каждая строка матрицы представляет уникальное слово. Если в исходном корпусе[1] документов, который передается в алгоритм обработки естественного языка, имеется 100 уникальных слов, тогда матрица прямого кодирования слов будет иметь 100 строк. Если в корпусе содержится 1000 уникальных слов, матрица будет содержать 1000 строк и т. д.

The bat sat on the cat.

слова

the	1	0	0	0	1	0
bat	0	1	0	0	0	0
on	0	0	0	1	0	0
⋮						
n_{unique_words}						

Рис. 2.4. Прямое кодирование слов, как в этом примере, преобладает в обработке естественного языка на основе традиционного машинного обучения

Ячейки в матрицах прямого кодирования содержат двоичные значения, то есть 0 или 1. Каждый столбец содержит не более одной единицы, а в остальных ячейках находятся нули; это означает, что матрицы прямого кодирования являются *разреженными*[2]. Единичные значения указывают на присутствие определенного слова (строки) в определенной позиции (столбец) в корпусе. Корпус, изображенный на рис. 2.4, содержит только шесть слов, пять из которых уникальны. Соответственно, матрица прямого кодирования нашего корпуса состоит из

[1] *Корпус* (от лат. «тело») — это совокупность всех документов («тело» языка), которые используются в качестве входных данных для данного приложения обработки естественного языка. В главе 11 мы будем использовать корпус из 18 классических книг. Далее в этой главе мы используем отдельный корпус из 25 000 обзоров фильмов. Примером корпуса гораздо большего объема может служить набор статей из Википедии. Еще более объемный корпус — набор всех общедоступных страниц в Интернете, представленный на сайте commoncrawl.org.

[2] Значения, отличные от нуля, редки (то есть разрежены) в разреженной матрице. *Плотные* матрицы, напротив, богаты информацией: обычно они содержат лишь несколько нулевых значений, если вообще содержат.

шести столбцов и пяти строк. Первое уникальное слово — `the` — встречается в первой и пятой позициях, на что указывают ячейки в первой строке матрицы, содержащие `1`. Второе уникальное слово в корпусе — `bat` — встречается только во второй позиции, поэтому ему соответствует значение `1` в ячейке на пересечении второй строки и второго столбца. Подобные матрицы прямого кодирования слов имеют очень простую структуру и прекрасно подходят для передачи в модель глубокого обучения (или в другие модели машинного обучения). Однако, как будет показано чуть ниже, из-за простоты и разреженности прямое кодирование имеет ограниченное применение в приложениях обработки естественного языка.

ВЕКТОРЫ СЛОВ

Векторные представления слов являются более информационно плотной альтернативой прямому кодированию. Представление в форме прямого кодирования хранит информацию только о местоположении слов, тогда как *векторы слов* (их также называют *векторными представлениями* или *представлением в пространстве векторов*) хранят информацию не только о местоположении, но и о значении слов[1]. Наличие дополнительной информации делает векторы слов более предпочтительными по множеству причин, о которых мы поговорим далее в этой главе. Однако главное преимущество заключается в том, что, подобно визуальным признакам, которые автоматически изучаются моделями компьютерного зрения на основе глубокого обучения, как рассказывалось в главе 1, векторы слов позволяют моделям NLP автоматически изучать лингвистические признаки[2].

Для иллюстрации механизма построения векторов слов на рис. 2.5 показан крошечный пример. Начиная с первого слова в корпусе и двигаясь вправо, пока не будет достигнуто последнее слово в корпусе, мы по очереди рассматриваем каждое слово как *целевое*. В конкретный момент, показанный на рис. 2.5, целевым является слово `word`. Следующим целевым словом будет `by`, затем `the`, затем `company` и т. д. Для каждого целевого слова мы рассматриваем его окружение —

---

[1] Строго говоря, представление в форме прямого кодирования тоже является «векторным представлением», потому что каждый столбец матрицы прямого кодирования является вектором, представляющим данное слово в конкретном месте. Однако в сообществе глубокого обучения термин «векторы слов» обычно используется для обозначения плотных представлений, рассматриваемых в этом разделе, то есть тех, которые получены с помощью word2vec, GloVe и других аналогичных алгоритмов.

[2] Как отмечалось в начале этой главы, такое понимание значения слова в контексте окружающих его слов было предложено Людвигом Витгенштейном (Ludwig Wittgenstein). Позже, в 1957 году, эту идею кратко выразил британский лингвист Дж. Р. Фертом (J. R. Firth) в своей знаменитой фразе: «You shall know a word by the company it keeps» («Смысл слова определяется по его окружению»). Firth J. (1957). «Studies in linguistic analysis». Oxford: Blackwell.

контекстные слова. В нашем примере мы используем окно контекста размером в три слова. Это означает, что в контекст целевого слова `word` входят три слова слева (`a`, `know` и `shall`) и три слова справа (`by`, `company` и `the`), то есть всего шесть контекстных слов[1]. После перехода к последующему целевому слову (`by`) окно контекста также сдвигается на одну позицию вправо, и из него исключаются слова `shall` и `by`, а включаются слова `word` и `it`.

Рис. 2.5. Небольшой пример, иллюстрирующий процесс построения векторов слов, лежащий в основе таких алгоритмов, как word2vec и GloVe, используемых для преобразования текста на естественном языке в векторы слов

Двумя наиболее популярными методами преобразования текстов на естественном языке в векторы слов являются *word2vec*[2] и *GloVe*[3]. Независимо от метода, главная цель при рассмотрении любого конкретного целевого слова состоит в том, чтобы точно предсказать его по контексту[4]. Увеличивая точность этих предсказаний, слово за словом в большом корпусе, мы постепенно выясняем, какие слова имеют тенденцию появляться в схожих контекстах, и назначаем им близкие координаты в векторном пространстве.

На рис. 2.6 показана модель векторного пространства. Пространство может иметь любое количество измерений, поэтому его можно назвать n-мерным векторным пространством. На практике, в зависимости от богатства корпуса и сложности решаемой задачи NLP, можно создать векторное пространство слов с десятками, сотнями и даже тысячами измерений. Как мы уже заметили, любому данному слову из корпуса (например, `king`) присваиваются определен-

---

[1] Математически проще и эффективнее не учитывать конкретный порядок контекстных слов, потому что порядок слов обычно несет незначительный объем дополнительной информации. Поэтому в тексте контекстные слова в скобках указаны в алфавитном порядке, который фактически является случайным.

[2] *Mikolov T.* et al. (2013). «Efficient estimation of word representations in vector space». *arXiv:1301.3781.*

[3] *Pennington J.* et al. (2014). «GloVe: Global vectors for word representations». Proceedings of the Conference on Empirical Methods in Natural Language Processing.

[4] Или, напротив, предсказать контекстные слова по заданному целевому слову. Подробнее об этом рассказывается в главе 11.

ные координаты в векторном пространстве. Скажем, местоположение слова `king` в 100-мерном пространстве определяется вектором, обозначим его v_{king}, состоящим из 100 чисел, которые определяют координаты слова `king` во всех доступных измерениях.

Человеческому мозгу трудно представить пространство, имеющее более трех измерений, поэтому модель на рис. 2.6 имеет только три измерения. Каждому слову из нашего корпуса присваиваются три числовые координаты в этом трехмерном пространстве, которые определяют его местоположение: x, y и z. В этом примере значение слова `king` представлено вектором v_{king}, состоящим из трех чисел. Если вообразить, что вектор v_{king} содержит координаты $x = -0.9$, $y = 1.9$ и $z = 2.2$[1], тогда, чтобы кратко описать эти координаты, можно использовать форму записи `[-0.9, 1.9, 2.2]`. Эта краткая форма записи вскоре пригодится, когда мы начнем выполнять арифметические операции над векторами слов.

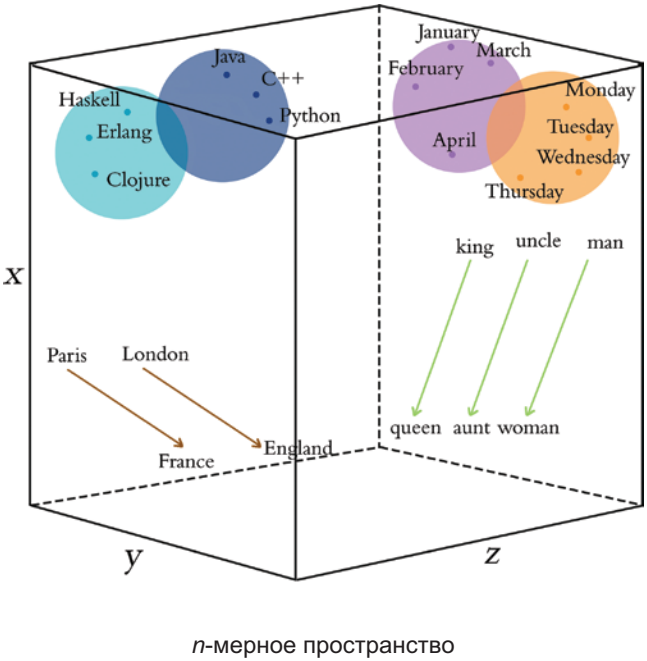


Рис. 2.6. Диаграмма, представляющая значения слов в трехмерном векторном пространстве

[1] В русском языке допустимо использование записи десятичных чисел как через точку, так и через запятую. В компьютерной литературе мы осознанно используем десятичную точку, чтобы не допускать разночтения в фрагментах кода и формульной части. (Ср. вариант «0, 0.0001, 0.25, 0.784, 0.9995», который читается и воспринимается лучше, чем «0, 0,0001, 0,25, 0,784, 0,9995»). — *Примеч. ред.*

Чем ближе два слова находятся в векторном пространстве[1], тем ближе они по смыслу, который определяется сходством контекстных слов, окружающих их в тексте на естественном языке. Предполагается, что синонимы и слова с типичными орфографическими ошибками — как схожие по значению — будут иметь почти идентичный контекст и, следовательно, почти идентичные координаты в векторном пространстве. Слова, которые используются в сходных контекстах, например обозначающих время, имеют тенденцию располагаться рядом друг с другом в векторном пространстве. На рис. 2.6 `Monday` (понедельник), `Tuesday` (вторник) и `Wednesday` (среда) представлены оранжевыми точками, расположенными в оранжевом круге-кластере, обозначающем дни недели. Между тем названия месяцев находятся в собственном фиолетовом кластере, который находится поблизости, но отличается от кластера дней недели; они оба относятся к понятию даты, но образуют отдельные подкластеры в более широком кластере *дат*. Аналогично можно ожидать, что названия языков программирования будут сгруппированы вместе в некоторой области в пространстве векторов слов, находящейся далеко от слов, обозначающих время, скажем, в верхнем левом углу. И снова здесь предполагается, что такие объектно-ориентированные языки программирования, как `Java`, `C++` и `Python`, образуют один подкластер, а поблизости находятся функциональные языки, такие как `Haskell`, `Clojure` и `Erlang`, образующие свой подкластер. Как вы увидите в главе 11, когда начнете сами конструировать векторные пространства слов, более общие термины, которые, тем не менее, передают конкретное значение — например, `created` (создание), `developed` (строительство) и `built` (сборка) — также располагаются в пространстве векторов слов так, что их позиции можно успешно использовать в задачах NLP.

АРИФМЕТИКА С ВЕКТОРАМИ СЛОВ

Примечательно, что направления в векторном пространстве являются эффективным средством предоставления информации об отношениях между конкретными словами, хранящимися в этом пространстве. Это потрясающее свойство[2]. Вернемся к нашему кубу на рис. 2.6. Коричневые стрелки представляют отношения между странами и их столицами. То есть если вычислить направление и расстояние между координатами слов `Paris` и `France`, а затем отложить точно такое же расстояние и в том же направлении от слова `London`, мы должны оказаться в окрестностях слова `England`. Аналогично можно рассчитать направление и расстояние между координатами слов `man` (мужчина) и `woman` (женщина).

---

[1] Измеряется евклидовым расстоянием — обычное линейное расстояние между двумя точками.

[2] Один из авторов книги — Джон — предпочитал описывать это свойство векторов слов, используя такие термины, как «офигенное» и «обалденное», но вынужден был обратиться к словарю, чтобы подобрать прилагательное, выглядящее более профессионально.

Это направление и расстояние в векторном пространстве будет представлять понятие пола (обозначено на рис. 2.6 зелеными стрелками). Если отложить это расстояние в том же направлении от какого-либо конкретного слова мужского рода, например `king` (король), `uncle` (дядя), мы должны попасть в координаты рядом с аналогичным словом женского рода — `queen` (королева), `aunt` (тетя).

$$v_{king} - v_{man} + v_{woman} = v_{queen}$$
$$v_{bezos} - v_{amazon} + v_{tesla} = v_{musk}$$
$$v_{windows} - v_{microsoft} + v_{google} = v_{android}$$

Рис. 2.7. Примеры арифметических операций с векторами слов

Побочным продуктом способности определять смысловые векторы (например, пол, отношение столица — страна), направленные от одного слова в векторном пространстве к другому, является возможность выполнения *арифметических операций с векторами слов*. Вот канонический пример: пусть есть вектор v_{king}, представляющий слово `king` (продолжая пример из предыдущего раздела, его местоположение описывается как `[-0.9, 1.9, 2.2]`), вычтем из него вектор, представляющий слово `man` (пусть v_{man} = `[-1.1, 2.4, 3.0]`), и прибавим вектор, представляющий слово `woman` (пусть v_{woman} = `[-3.2, 2.5, 2.6]`). В результате мы должны получить местоположение рядом с вектором, представляющим слово `queen`. Чтобы было понятнее, выполним эти арифметические операции с конкретными измерениями и оценим получившейся вектор v_{queen}:

$$\begin{aligned} x_{queen} &= x_{king} - x_{man} + x_{woman} = -0.9 + 1.1 - 3.2 = -3.0 \\ y_{queen} &= y_{king} - y_{man} + y_{woman} = 1.9 - 2.4 + 2.5 = 2.0 \\ z_{queen} &= z_{king} - z_{man} + z_{woman} = 2.2 - 3.0 + 2.6 = 1.8. \end{aligned} \tag{2.1}$$

После объединения всех трех измерений ожидается, что вектор v_{queen} окажется в точке около `[-3.0, 2.0, 1.8]`.

На рис. 2.7 показаны другие интересные примеры арифметики в векторном пространстве слов, полученном в результате обучения на большом корпусе, взятом из Интернета. Как вы увидите в главе 11, наличие этих смысловых количественных отношений между словами в векторном пространстве является надежной отправной точкой для моделей глубокого обучения в приложениях NLP.

WORD2VIZ

Чтобы получить еще более полное представление о векторах слов, перейдите по ссылке bit.ly/word2viz. На рис. 2.8 показана начальная страница инструмента word2viz для интерактивного исследования векторов слов. Оставьте

в раскрывающемся списке справа вверху значение Gender analogies (Аналогии с полами), попробуйте добавить *пары* новых слов в разделе Modify words (Изменить слова). Если добавить пары соответствующих гендерных слов, таких как `princess` (принцесса) и `prince` (принц), `duchess` (герцогиня) и `duke` (герцог) или `businesswoman` (деловая женщина) и `businessman` (бизнесмен), вы обнаружите, что они попадают во вполне определенные места.

Разработчик инструмента word2viz Джулия Базинска (Julia Bazińska) сжала 50-мерное векторное пространство слов до двухмерного, чтобы получить возможность отображать векторы в системе координат *xy*[1]. В настройках по умолчанию Базинска ориентировала ось *x* в направлении от слова `she` (она) к слову `he` (он) для представления пола, а ось *y* — от понятия «люди» к понятию «царственные особы», ориентируясь на слова `woman` (женщина) и `queen` (королева). Отображаемые слова, помещенные в векторное пространство посредством обучения на наборе данных естественного языка, состоящем из 6 миллиардов образцов использования 400 000 уникальных слов[2], расположились относительно двух осей в зависимости от их значения. Чем ближе понятие к царственным особам (к слову `queen`), тем выше на графике находится соответствующее слово; указания на принадлежность к женскому полу оказались слева, а к мужскому — справа.

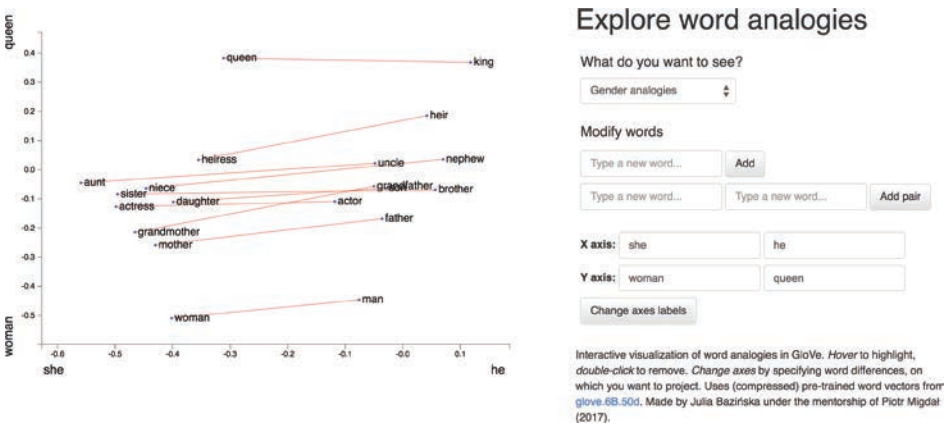


Рис. 2.8. Начальная страница инструмента word2viz для интерактивного исследования векторов слов

Удовлетворив свое первое любопытство с гендерными аналогиями в word2viz, поэкспериментируйте с другими вариантами организации векторного про-

[1] О том, как уменьшить размерность векторного пространства для целей визуализации, мы расскажем в главе 11.

[2] Если говорить точнее, то 400 000 *лексем*; о различиях между словами и лексемами мы поговорим позже.

странства слов. Выбрав значение Adjectives analogies (Аналогии прилагательных) в раскрывающемся списке What do you want to see? (Что вы хотите увидеть?), можно, например, добавить слова `small` (маленький) и `smallest` (наименьший). Также попробуйте слова `nice` (милый) и `nicer` (милее), а затем снова `small` (маленький) и `big` (большой). Выберите в списке Numbers say-write analogies (Аналогии произношения — написания чисел) и поэкспериментируйте, изменив масштаб оси x от `3` до `7`.

При желании можно создать свой график в word2viz, выбрав представление Empty (Пусто). Какой вектор выбрать — решать вам, но, возможно, будет интересно посмотреть, как выглядят отношения страна — столица, о которых упоминалось выше, когда мы рассматривали рис. 2.6. Для этого выберите для оси x граничные значения `west` (запад) и `east` (восток), а для оси y — `city` (город) и `country` (страна). В эту систему координат прекрасно вписываются пары слов: `london`–`england`, `paris`–`france`, `berlin`–`germany` и `beijing`–`china`.

С одной стороны, word2viz дает замечательную возможность получить более полное представление о векторах слов, с другой стороны, с его помощью можно исследовать сильные и слабые стороны данного векторного пространства слов. Например, выберите в раскрывающемся списке What do you want to see? (Что вы хотите увидеть?) пункт Verb tenses (Времена глаголов), а затем добавьте слова `lead` (ведет) и `led` (вел). После этого вы обнаружите, что координаты, назначенные этим словам в данном векторном пространстве, отражают существующие гендерные стереотипы, присутствовавшие в данных, на которых обучалось векторное пространство. После переключения на представление Jobs (Работа) эта гендерная предвзятость становится еще более очевидной. Можно с уверенностью сказать, что любой большой набор данных естественного языка будет страдать некоторой смещенностью, преднамеренно или нет. В настоящее время активно исследуются возможности уменьшения смещенности в векторах слов[1]. Не забывайте, что она может присутствовать и в ваших данных, поэтому старайтесь тестировать свои приложения NLP в разных ситуациях для разных групп пользователей и проверяйте соответствие результатов.

ЛОКАЛЬНЫЕ И РАСПРЕДЕЛЕННЫЕ ПРЕДСТАВЛЕНИЯ

Получив интуитивное понимание векторов слов, попробуем сопоставить их с представлением в прямом кодировании (рис. 2.4), которое уже давно используется в мире NLP. В целом можно сказать, что в векторном представлении смысл слов *распределен* в n-мерном пространстве, то есть значения слов *размазаны* по векторному пространству. В прямом кодировании смысл слов хранится

[1] Например: *Bolukbasi T.* et al. (2016). «Man is to computer programmer as woman is to homemaker? Debiasing word embeddings». *arXiv:1607.06520*; *Caliskan A.* et al. (2017). @Semantics derived automatically from language corpora contain human-like biases». *Science 356: 183–6*; *Zhang B.* et al. (2018). «Mitigating unwanted biases with adversarial learning». *arXiv:1801.07593*.

локально. Это представление хранит информацию о данном слове дискретно, в пределах одной строки часто чрезвычайно разреженной матрицы.

Чтобы более полно охарактеризовать различия между локальным прямым кодированием и распределенным векторным представлением, в табл. 2.1 приводится их сравнение по ряду атрибутов. Во-первых, прямому кодированию не хватает нюансов; это простой набор двоичных флагов. Векторные представления, напротив, чрезвычайно богаты нюансами: информация о словах в них распределена по непрерывному числовому пространству. Это многомерное пространство обладает по сути бесконечными возможностями для представления отношений между словами.

Таблица 2.1. Сравнение атрибутов локального прямого кодирования и распределенного векторного представления

Прямое кодирование	Векторное представление
Мало деталей	Богато нюансами
Классификация должна выполняться вручную	Все делается автоматически
Плохо справляется с новыми словами	Легко внедряет новые слова
Зависит от субъективных решений	Определяется только исходными данными
Отсутствует возможность определить сходство слов	Сходство слов = близость в пространстве

Во-вторых, использование прямого кодирования на практике часто требует вручную выполнить трудоемкую классификацию, включая создание словарей и других специализированных лингвистических баз данных[1]. Такая классификация не требуется для векторных представлений, которые создаются полностью автоматически, только на основе исходных данных.

В-третьих, представления в форме прямого кодирования плохо справляются с новыми словами. Для нового слова требуется добавить строку в матрицу, повторно проанализировать отношения между существующими строками и изменить код, возможно, с добавлением ссылок на внешние источники информации. При использовании векторных представлений новые слова можно добавлять повторным обучением векторного пространства на корпусе, включающем примеры слов в их естественном контексте. В этом случае новое слово получит свой n-мерный вектор. Первоначально обучающих данных, включающих новое слово, может оказаться немного, из-за чего вектор может недостаточно точно соответствовать фактическому местоположению слова в n-мерном про-

[1] Как, например, база данных WordNet (wordnet.princeton.edu), которая описывает как синонимы, так и гиперонимы (описывают отношения вида «является», то есть слово «мебель», например, является гиперонимом слова «стул»).

странстве, но расположение всех существующих слов остается неизменным, и модель будет продолжать нормально функционировать. Со временем, когда появится больше примеров нового слова в корпусе, точность представления его местоположения в векторном пространстве увеличится[1].

В-четвертых, как вытекает из двух предыдущих пунктов, использование прямого кодирования часто предполагает субъективную интерпретацию смысла. Именно потому нередко требуется определять правила кодирования и привлекать справочные базы данных, которые разрабатываются относительно небольшими группами специалистов. Между тем смысл слов в векторных представлениях определяется исходными данными[2].

В-пятых, прямое кодирование не учитывает сходства слов: похожие слова, такие как «диван» и «софа», представлены точно так же, как не связанные, например «диван» и «кот». В векторном представлении, напротив, сходство слов учитывается достаточно полно: как упоминалось выше в описании рис. 2.6, чем более похожи слова, тем ближе друг к другу они расположены в векторном пространстве.

ЭЛЕМЕНТЫ ЕСТЕСТВЕННОГО ЯЗЫКА

До сих пор мы рассматривали только один элемент естественного языка: *слово*. Однако слова сами состоят из отдельных элементов и одновременно являются составными частями более абстрактных и сложных языковых элементов. Сначала мы рассмотрим языковые элементы, составляющие слова, а затем проследуем по схеме, изображенной на рис. 2.9. Для каждого элемента мы обсудим, как он обычно кодируется с точки зрения традиционного машинного обучения, а также с точки зрения глубокого обучения. По мере движения вперед отметим, что распределенные векторные представления, используемые в глубоком обучении, являются текучими и гибкими, тогда как представления в традиционном машинном обучении — локальными и жесткими (табл. 2.2).

Фонология — раздел лингвистики, изучающий структуру *звукового строя* языка. В каждом языке есть определенный набор *фонем* (звуков), составляющих слова. В традиционном машинном обучении используется подход, основанный на кодировании сегментов, воспринимаемых на слух, в виде конкретных фонем из

---

[1] Сопутствующая проблема, не рассматриваемая здесь, возникает, когда алгоритм NLP встречает слово, не включенное в обучающий набор данных. Эта проблема *отсутствия в словаре* затрагивает и представление в форме прямого кодирования, и векторное представление. Уже есть решения, такие как библиотека *fastText* в Facebook, позволяющие обойти эту проблему за счет анализа частей слова, но их обсуждение выходит за рамки этой книги.

[2] Тем не менее, как было подмечено выше во врезке, они могут страдать проблемой смещенности, обусловленной смещенностью самих исходных данных.

числа доступных в языке. В глубоком обучении модель прогнозирует фонемы по признакам, автоматически извлекаемым из исходной речевой последовательности, а затем представляет их в векторном пространстве. В этой книге мы работаем с естественным языком только в текстовом формате, но описанные здесь методы можно применить и непосредственно к речевым данным.

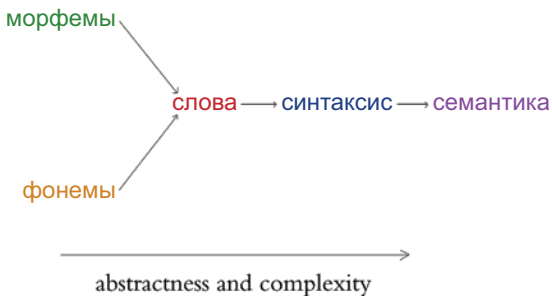


Рис. 2.9. Связь между элементами естественного языка. Элементы слева служат строительными блоками для элементов справа. Чем дальше вправо, тем более абстрактными становятся элементы и, следовательно, тем сложнее их моделировать в приложениях NLP

Морфология изучает формы слов. Кроме фонем в каждом языке есть свой набор *морфем*, которые являются наименьшими единицами языка, наделенными некоторым смыслом. Например, три морфемы, `out`, `go` и `ing`, объединяются и образуют слово `outgoing` (исходящий). В традиционном машинном обучении они идентифицируются в тексте по списку всех морфем в данном языке. В глубоком обучении модель должна предсказывать появление определенных морфем. Иерархически более глубокие слои искусственных нейронов могут затем комбинировать несколько векторов (например, три морфемы — `out`, `go` и `ing`) в один, представляющий слово.

Таблица 2.2. Представление элементов естественного языка в традиционном и глубоком машинном обучении

Представление	Традиционное машинное обучение	Глубокое обучение	Только для речи
Фонология	Все фонемы	Векторы	Да
Морфология	Все морфемы	Векторы	Нет
Слова	Прямое кодирование	Векторы	Нет
Синтаксис	Правила составления фраз	Векторы	Нет
Семантика	Лямбда-исчисление	Векторы	Нет

Фонемы (используются при изучении речи) и морфемы (используются при изучении текста) объединяются в *слова*. В этой книге, используя данные естественного языка, мы будем работать на уровне слов. Этот выбор сделан по четырем причинам. Во-первых, легко определить, что такое слова, и все мы знаем, как они выглядят. Во-вторых, естественный язык легко разбить на слова, если использовать процесс, называемый *лексемизацией*[1] (tokenization); мы используем его в главе 11. В-третьих, слова являются наиболее изученным уровнем естественного языка, особенно это относится к глубокому обучению, поэтому есть возможность применять к ним самые передовые методики. В-четвертых, и это, пожалуй, наиболее важно, векторы слов лучше подходят для моделей NLP, которые мы будем конструировать и которые отличаются функциональностью, эффективностью и точностью. В предыдущем разделе мы перечислили недостатки локальных представлений в форме прямого кодирования, которые преобладают в традиционном машинном обучении, и сравнили их с векторными представлениями, используемыми в моделях глубокого обучения.

Слова объединяются и формируют *синтаксис*. Синтаксис и морфология вместе составляют грамматику языка. Синтаксис определяет порядок следования слов во фразах и словосочетаний в предложениях для передачи смысла, понятного носителям данного языка. В традиционном машинном обучении фразы разбиваются на отдельные формальные лингвистические категории[2]. В глубоком обучении используются векторы (сюрприз!). Каждое слово и каждая фраза во фрагменте текста могут быть представлены вектором в n-мерном пространстве, а слои искусственных нейронов могут объединять слова в фразы.

Семантика — самый абстрактный из элементов естественного языка, представленных на рис. 2.9 и в табл. 2.2; она определяет *смысл* предложений. Смысл выводится из базовых языковых элементов, таких как слова и фразы, а также из общего контекста, окружающего фрагмент текста. Определение смысла — сложная задача, потому что решение о том, как следует понимать текст — буквально или как юмористическое и саркастическое замечание, — может зависеть от тонких контекстуальных различий и меняющихся культурных норм. Традиционное машинное обучение не отражает нечеткость языка (например, сходство родственных слов или фраз), поэтому имеет ограниченные возможности в определении семантического значения. В глубоком обучении векторы снова приходят на помощь. Они могут представлять не только каждое слово и фразу в тексте, но и каждое логическое выражение. По аналогии с уже рассмотренными языковыми элементами слои искусственных нейронов могут рекомбинировать векторы составляющих элементов и вычислять семантические векторы, составляя нелинейные комбинации векторов фраз.

---

[1] Фактически лексемизация — это процесс определения границ слов по таким символам, как запятые, точки и пробелы.

[2] Эти категории имеют такие названия, как «именное словосочетание» и «глагольное словосочетание».

GOOGLE DUPLEX

Одним из наиболее заметных примеров реализации обработки естественного языка на основе глубокого обучения является технология Google Duplex, которая была представлена в мае 2018 года на конференции разработчиков Google I/O. Генеральный директор поискового гиганта Сундар Пичаи (Sundar Pichai) привел зрителей в восторг, продемонстрировав, как помощник Google звонит по телефону в китайский ресторан и бронирует столик. По залу прокатился возглас удивления, когда зрители услышали, насколько естественно система Duplex ведет диалог. Он освоил некоторые особенности человеческого диалога, изобилующего такими словами, как «эх», «угу», «хм-м». Кроме того, качество телефонной связи было весьма посредственным, и человек на том конце линии говорил с сильным акцентом; но Duplex прекрасно справилась с задачей и забронировала столик.

Конечно, это была всего лишь демонстрация, проводившаяся даже не в живую, но широта приложений глубокого обучения, которые нужно объединить для создания этой технологии, поразительна. Представьте, как выглядит поток информации между участниками разговора (Duplex и ресторатор): Duplex должна включать сложный алгоритм распознавания речи, способный обрабатывать поток звуков в режиме реального времени, широкий спектр акцентов, разное качество телефонной связи, а также отсекать фоновый шум[1].

После достоверной расшифровки и транскрибирования речи человека модель NLP должна обработать предложение и определить, что оно означает. Кроме того, цель состояла в том, чтобы не дать собеседнику возможности догадаться, что он разговаривает с компьютером, а значит, модель должна соответствующим образом модулировать свою речь. Кроме того, люди часто отвечают сложными составными предложениями, понять которые — очень непростая задача для компьютера, например:

> *«На завтра ничего нет, но есть на следующий день и в четверг, в любое время до восьми. Хотя нет... В четверг в семь тоже ничего. Но, если хотите, можно после восьми».*

Эта фраза плохо структурирована — вы никогда не напишете так в электронном письме, но в естественном разговоре такие поправки и замены на лету происходят регулярно, и система Duplex должна была справиться с ними.

---

[1] Эта проблема известна как «проблема шумной вечеринки» или, если использовать технически более строгое название, «распознавание речи разных участников». Люди способны решать эту проблему с рождения, прекрасно справляясь с задачей выделения отдельных голосов из какофонии звуков без четких инструкций о том, как это сделать. Машины обычно не способны справиться с ней, хотя различные группы исследователей предлагали решения. Например: *Simpson A.* et al. (2015). «Deep karaoke: Extracting vocals from musical mixtures using a convolutional deep neural network». *arXiv:1504.04658*; *Yu D.* et al. (2016). «Permutation invariant training of deep models for speaker-independent multi-talker speech separation». *arXiv:1607.00325*.

Транскрибировав речь и определив смысл предложения, Duplex конструирует ответ. В нем она должна либо запросить дополнительную информацию, если человек был недостаточно точен или его ответы были неудовлетворительными, либо подтвердить бронирование. Модель NLP генерирует ответ в текстовой форме, поэтому необходим также механизм преобразования текста в речь (Text-To-Speech, TTS).

Для синтеза речи в Duplex используется комбинация из Tacotron[1] и WaveNet[2], а также классического «конкатенативного» механизма преобразования текста в речь[3]. Здесь система входит в «зловещую долину»[4]: ресторатор слышит вовсе не человеческий голос. WaveNet может генерировать полностью синтетические сигналы, используя глубокую нейронную сеть, обученную на реальных звуках, издаваемых людьми. Модель Tacotron отображает последовательности слов в соответствующие ряды *звуковых признаков*, которые фиксируют такие тонкости человеческой речи, как высота, скорость, интонация и даже произношение. Эти признаки затем передаются в модель WaveNet, которая синтезирует фактические звуки. Вся эта система способна воспроизводить естественный голос с правильной интонацией, эмоциями и акцентом. В более или менее типичные моменты разговора используется простой конкатенативный механизм TTS (вставляющий фрагменты из записей его *собственного* «голоса»), который менее требователен к вычислительным ресурсам. Вся эта конструкция динамически переключается между разными моделями по мере необходимости.

Перефразируя Джерри Магуайра (Jerry Maguire), все начинается с первой фразы «Алло». Система распознавания речи, модели NLP и механизм TTS работают согласованно с момента ответа на вызов, и с этого же момента Duplex приступает к решению сложнейшей задачи. Управляет всеми этими взаимодействиями глубокая нейронная сеть, которая специализируется на обработке информации, поступающей последовательно[5]. Она следит за ходом диалога и подает различные данные в соответствующие модели.

---

[1] bit.ly/tacotron

[2] bit.ly/waveNet

[3] Конкатенативные механизмы преобразования текста в речь используют обширные базы данных предварительно записанных слов и фрагментов, которые можно объединять для получения предложений. Это распространенный и довольно простой подход, но он дает речь, непохожую на естественную, и не способен адаптировать скорость и интонацию. Например, он не способен модулировать слова, чтобы они звучали как вопрос.

[4] Зловещая долина — опасная ситуация, когда робот или другой объект, выглядящий или действующий примерно как человек (но не в точности как настоящий), вызывает чувства дискомфорта и страха у наблюдателей. Дизайнеры продуктов стараются избежать попадания в зловещую долину, зная, что люди доброжелательно реагируют на модели, которые либо очень хорошо имитируют человека, либо вообще не стараются быть на него похожими. (https://ru.wikipedia.org/wiki/Зловещая_долина. — *Примеч. пер.*)

[5] Она называется *рекуррентной нейронной сетью*. Сети этого типа мы обсудим в главе 11.

Как нетрудно догадаться, Google Duplex — это сложная система моделей глубокого обучения, работающих вместе для поддержания устойчивого диалога по телефону. В настоящее время Duplex имеет весьма ограниченные возможности: она способна лишь запланировать встречу и забронировать столик. Она не может вести общие разговоры. Поэтому, несмотря на то что Duplex представляет значительный шаг вперед в развитии искусственного интеллекта, нам предстоит еще много работы.

ИТОГИ

В этой главе мы познакомились с использованием глубокого обучения для обработки естественного языка. Мы особо подчеркнули способность моделей глубокого обучения автоматически извлекать из данных наиболее подходящие признаки, избавляющую нас от необходимости вручную конструировать представления языка в форме прямого кодирования. Вместо этого приложения NLP, основанные на глубоком обучении, используют векторные представления, которые достаточно точно фиксируют мелкие детали значений слов, что увеличивает точность модели.

В главе 11 мы сконструируем приложение NLP, используя искусственные нейронные сети для обработки данных на естественном языке и получения суждений об этих данных. В таких «сквозных» моделях глубокого обучения начальные слои создают векторы слов, плавно перетекающие в более глубокие специализированные слои, в том числе и обладающие «памятью». Эти архитектуры моделей подчеркивают силу и простоту глубокого обучения при использовании векторов слов.

3

МАШИННОЕ ИСКУССТВО

В этой главе мы познакомимся с некоторыми идеями, позволяющими моделям глубокого обучения *творить*, и кому-то эти идеи могут показаться парадоксальными. Как сказал философ Алва Ноэ (Alva Noë) из Калифорнийского университета в Беркли, «искусство может помочь нам написать лучшую картину человеческой природы»[1]. Но как машины могут творить? Или, иначе говоря, можно ли рассматривать творения машин как настоящее искусство? Другая интерпретация, которая нам нравится больше всего, заключается в том, что такие творения действительно являются искусством, а программисты — художниками, использующими модели глубокого обучения в качестве кистей. Мы не единственные, кто считает эти работы полноценным искусством: некоторые картины, нарисованные генеративно-состязательными сетями (Generative Adversarial Network, GAN), были проданы по цене 400 000 долларов за штуку[2].

В этой главе мы рассмотрим высокоуровневые понятия, лежащие в основе GAN, и примеры художественных работ, которые они могут создавать. Мы проведем связь между скрытыми пространствами в GAN и пространствами векторов слов из главы 2, а также рассмотрим модель глубокого обучения, которую можно использовать как автоматизированный инструмент для улучшения качества фотографий. Но прежде давайте выпьем...

НОЧНАЯ ПЬЯНКА

На первом этаже в здании Google в Монреале находится бар под названием «Les 3 Brasseurs», которое переводится с французского как «Три пивовара». Это было третьесортное питейное заведение, когда в 2014 году Ян Гудфеллоу, будучи аспирантом в известной лаборатории Йошуа Бенжио (рис. 1.10), при-

[1] *Noë A.* (2015, October 5). «What art unveils». The New York Times.
[2] *Cohn G.* (2018, October 25). «AI art at Christie's sells for $432.500». The New York Times.

думал алгоритм получения реалистичных изображений[1], который Ян Лекун (см. рис. 1.9) назвал «самым важным» прорывом в глубоком обучении[2].

Друзья Гудфеллоу описали ему *генеративную модель*, над которой они работали, — вычислительную модель, целью которой было создание чего-то нового, будь то цитата в стиле Шекспира, музыкальная мелодия или произведение абстрактного изобразительного искусства. В данном конкретном случае друзья пытались разработать модель, способную генерировать фотореалистичные изображения, такие как портреты людей. Чтобы реализовать такую модель с использованием методов традиционного машинного обучения (рис. 1.12), инженерам-разработчикам потребовалось не только классифицировать и аппроксимировать важные индивидуальные особенности лиц, такие как глаза, нос и рот, но и точно оценить, как эти признаки должны располагаться относительно друг друга. До того момента полученные результаты не выглядели впечатляющими. Сгенерированные изображения получались слишком расплывчатыми или на них отсутствовали такие важные элементы, как нос или уши.

Возможно, благодаря паре кружек пива, усиливших его творческий потенциал[3], Гудфеллоу предложил революционную идею: использовать модель глубокого обучения, в которой две искусственные нейронные сети состязаются друг с другом. Как показано на рис. 3.1, одна из этих глубоких нейросетей должна быть запрограммирована на создание подделок, а вторая должна действовать

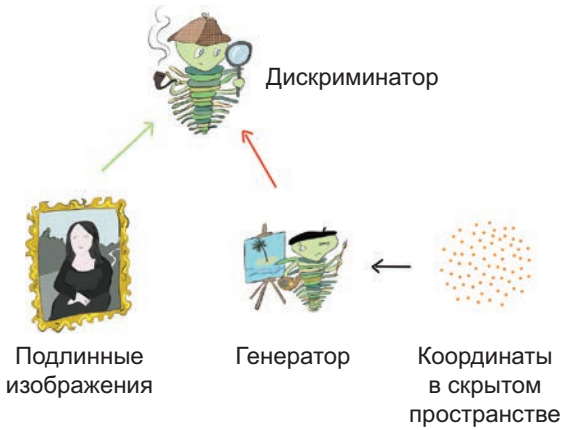


Рис. 3.1. Схематическое представление генеративно-состязательной сети (GAN). Настоящие изображения, а также подделки, созданные генератором, передаются дискриминатору, который старается отличить подлинные изображения от подделок. Оранжевое облако представляет скрытое «руководящее» пространство (рис. 3.4) для фальсификатора. Это пространство может быть случайным (как это обычно бывает при обучении сети; см. главу 12) либо выборочным (при исследовании после обучения, как на рис. 3.3)

[1] *Giles M.* (2018, February 21). «The GANfather: The man who's given machines the gift of imagination». MIT Technology Review.

[2] *LeCun Y.* (2016, July 28). «Quora». *bit.ly/DLbreakthru*

[3] *Jarosz A.* et al. (2012). «Uncorking the muse: Alcohol intoxication facilitates creative problem solving». Consciousness and Cognition, 21, 487–93.

как детектив и стараться отличать подделки от реальных изображений (которые будут предоставлены отдельно). Эти состязающиеся сети глубокого обучения должны подталкивать друг друга к развитию: по мере того как *генератор* будет все лучше справляться с созданием фальшивых изображений, *дискриминатор* будет надежнее их идентифицировать, вследствие чего генератор будет вынужден учиться производить еще более убедительные подделки, и т. д. Этот цикл в конечном итоге приведет к созданию новых убедительных образов в стиле настоящих обучающих изображений, будь то лица или что-то еще. Но самое замечательное в том, что подход, предложенный Гудфеллоу, избавляет от необходимости вручную проектировать признаки для генеративной модели. Как уже рассказывалось при обсуждении вопросов компьютерного зрения (в главе 1) и обработки естественного языка (в главе 2), модели глубокого обучения автоматически определяют наиболее подходящие признаки.

Друзья Гудфеллоу сомневались, что предложенный им подход сработает. Поэтому, когда он вернулся домой и обнаружил, что его подруга спит, он сел за компьютер и допоздна работал над созданием модели с двумя нейронными сетями. Модель заработала с первого раза, и в результате родилось целое семейство генеративно-состязательных сетей глубокого обучения!

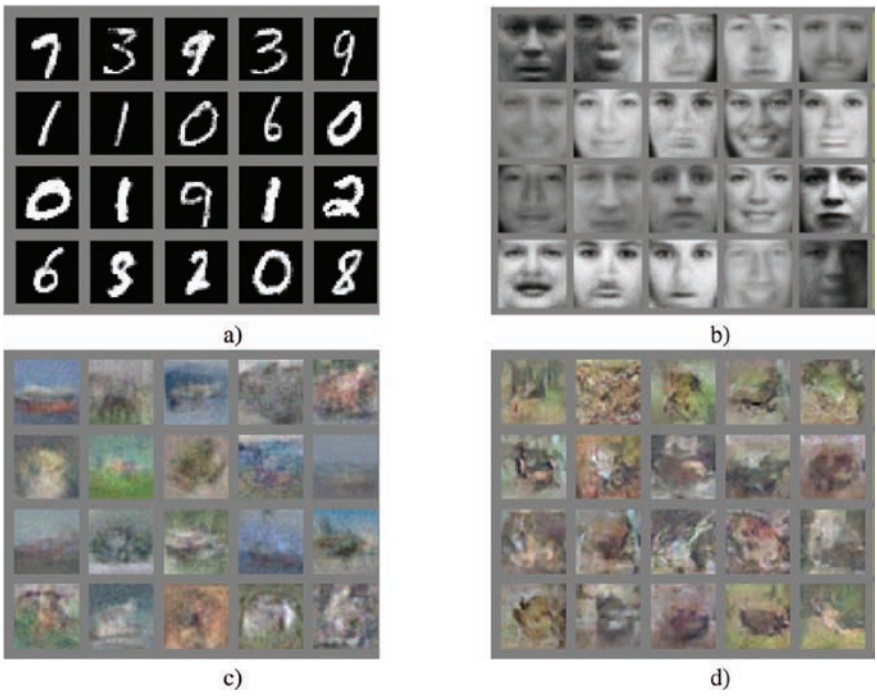


Рис. 3.2. Результаты, представленные Яном Гудфеллоу и его коллегами в статье с описанием генеративно-состязательных сетей в 2014 году

В том же году Гудфеллоу и его коллеги поведали миру о генеративно-состязательных сетях на престижной конференции по системам нейронной обработки информации (Neural Information Processing Systems, NIPS)[1]. Некоторые из их результатов показаны на рис. 3.2. Их сети GAN создали новые изображения, обучаясь на: (a) рукописных цифрах[2], (b) фотографиях человеческих лиц[3] и (c) и (d) фотографиях объектов из десяти разных классов (например, самолеты, автомобили, собаки)[4]. Изображения, получившиеся в (c), заметно менее четкие, чем в (d), потому что в последнем случае были задействованы слои, специализированные для компьютерного зрения (*сверточные* слои[5]), тогда как в сети для случая (c) использовались более универсальные слои[6].

АРИФМЕТИКА ИЗОБРАЖЕНИЙ НЕСУЩЕСТВУЮЩИХ ЛЮДЕЙ

Следуя предложениям Гудфеллоу, группа исследователей во главе с американским инженером по машинному обучению Алеком Рэдфордом (Alec Radford) определила архитектурные ограничения для генеративно-состязательных сетей, обеспечивающие наибольшую реалистичность созданных изображений. На рис. 3.3 представлены некоторые примеры портретов несуществующих людей, «нарисованные» *глубокими сверточными* сетями GAN[7]. В своей работе Рэдфорд с коллегами продемонстрировали интерполяцию и арифметику со *скрытым пространством*, связанным с GAN. Для начала определим, что такое скрытое пространство, а затем перейдем к интерполяции и арифметике с ним.

Скрытое пространство портретов на рис. 3.4 чем-то напоминает векторное пространство слов на рис. 2.6. Оказывается, скрытое пространство генеративно-состязательной сети и векторное пространство слов имеют три основных сходства. Во-первых, несмотря на то что для простоты на рис. 3.4 показано только три измерения, на самом деле скрытые пространства являются n-мерными и обычно имеют до нескольких сотен измерений. Например, скрытое пространство GAN, которое вы создадите в главе 12, будет иметь $n = 100$. Во-вторых, чем ближе две точки находятся в скрытом пространстве, тем более похожи изображения, которые они представляют. И в-третьих, перемещение в скрытом пространстве

---

[1] *Goodfellow I.* et al. (2014). «Generative adversarial networks». *arXiv:1406.2661.*

[2] Из классического набора данных MNIST, который мы используем во второй части книги.

[3] Из базы данных Toronto Face исследовательской группы Хинтона (рис. 1.16).

[4] Набор данных CIFAR-10, названный в честь Канадского института перспективных исследований (Canadian Institute for Advanced Research), поддержавшего его создание.

[5] Более подробно с этими слоями мы познакомимся в главе 10.

[6] Полносвязанные слои, которые будут представлены в главе 4 и подробно описаны в главе 7.

[7] *Radford A.* et al. (2016). «Unsupervised representation learning with deep convolutional generative adversarial networks». *arXiv:1511.06434v2.*

в любом конкретном направлении может соответствовать постепенному изменению представляемого понятия, такого как возраст или пол.

Рис. 3.3. Пример арифметики со скрытым пространством из работы Редфорда и др. (2016)

Выбрав две точки, расположенные далеко друг от друга вдоль некоторой *n*-мерной оси, представляющей возраст, и интерполируя точки между ними, можно получить изображения лица одного и того же человека (несуществующего), постепенно становящегося все старше и старше[1]. На схеме скрытого пространства (рис. 3.4) такая ось «возраста» изображена фиолетовым. Чтобы увидеть интерполяцию в скрытом пространстве действующей сети GAN, найдите в статье Рэдфорда и его коллег, например, описание плавного поворота «угла обзора» искусственных спален. С современным состоянием дел в развитии генеративно-состязательных сетей можно ознакомиться в видеоролике по адресу *bit.ly/InterpCeleb*. Он был создан исследователями из компании Nvidia, производящей видеокарты, и представляет собой захватывающую интерполяцию, созданную с помощью высококачественных портретных «фотографий» знаменитостей[2,3].

---

[1] Техническое замечание: как и в случае с векторными пространствами, такая ось «возраста» (или любое другое направление в скрытом пространстве, представляющее некоторую значимую характеристику) может быть ортогональна ко всем *n* измерениям, составляющим систему координат *n*-мерного пространства. Мы обсудим этот вопрос в главе 11.

[2] *Karras T.* et al. (2018). «Progressive growing of GANs for improved quality, stability, and variation». Proceedings of the International Conference on Learning Representations.

[3] Чтобы попробовать свои силы в различении лиц, реальных и созданных генеративно-состязательной сетью, посетите сайт *whichfaceisreal.com*.

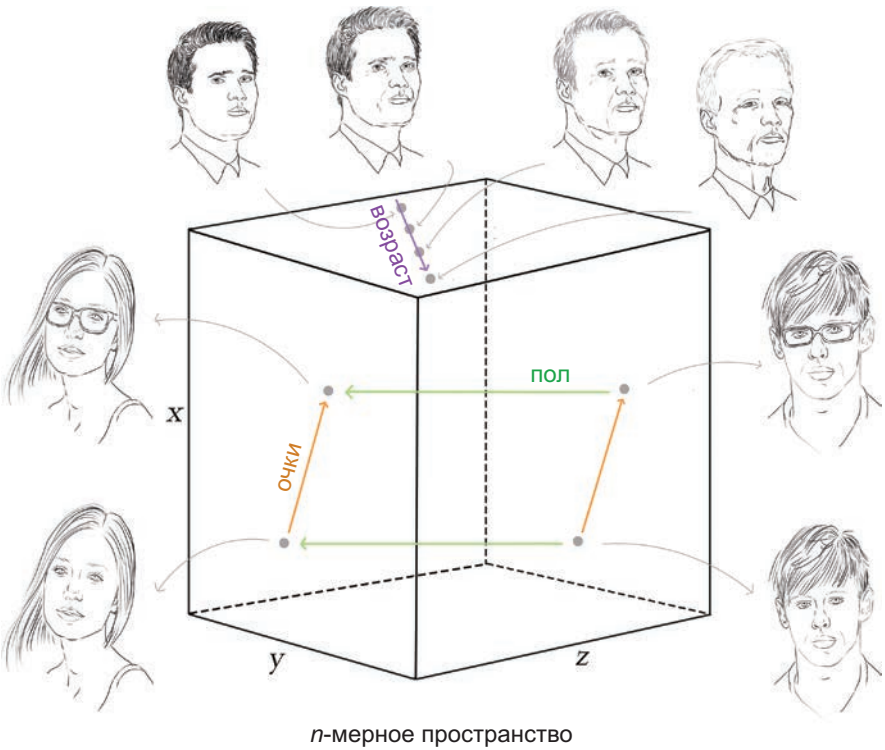


Рис. 3.4. Схема скрытого пространства, образованного генеративно-состязательной сетью (GAN). Направление вдоль фиолетовой стрелки соответствует увеличению возраста человека. Зеленая стрелка обозначает пол, а оранжевая — наличие очков

Теперь сделаем еще шаг и посмотрим, как можно выполнять арифметические операции с изображениями, полученными из скрытого пространства генеративно-состязательной сети. Чтобы выбрать точку в скрытом пространстве, нужно определить координаты ее местоположения. Полученный в результате вектор подобен векторам слов, описанным в главе 2. Точно так же, как в случае с векторами слов, к векторам изображений можно применять арифметические операции и перемещаться через скрытое пространство с учетом семантики направлений. На рис. 3.3 показаны примеры арифметики скрытого пространства из статьи Рэдфорда и его коллег. Начиная с точки в скрытом пространстве генеративно-состязательной сети, представляющей мужчину в очках, вычитая точку, представляющую мужчину *без* очков, и добавляя точку, представляющую *женщину* без очков, получается точка, находящаяся в скрытом пространстве рядом с изображениями женщин *в* очках. Схема на рис. 3.4 показывает, что скрытое пространство (подобно векторному пространству слов) хранит отношения между характеристиками, благодаря чему упрощается арифметика с точками в скрытом пространстве.

ПЕРЕДАЧА СТИЛЯ: ПРЕОБРАЗОВАНИЕ ФОТОГРАФИЙ В ИЗОБРАЖЕНИЯ В СТИЛЕ МОНЕ (И НАОБОРОТ)

Одним из наиболее магических применений GAN является *передача стиля*. Чжу (Zhu), Пак (Park) и их коллеги из лаборатории искусственного интеллекта в Беркли (Berkeley Artificial Intelligence Research, BAIR) представили новую версию GAN[1], которая демонстрирует потрясающие примеры передачи стиля, как показано на рис. 3.5. Алексей Эфрос (Alexei Efros), один из соавторов статьи, сделал фотографии во время отпуска во Франции, и исследователи использовали свою сеть CycleGAN, чтобы добавить в эти фотографии стили, свойственные художнику-импрессионисту Клоду Моне, голландскому художнику XIX века Винсенту Ван Гогу, а также традиционный стиль японских художников Укиё-е. По адресу bit.ly/cycleGAN вы найдете примеры обратного преобразования (картин Моне в фотореалистичные изображения), а также:

- летние пейзажи, преобразованные в зимние и наоборот;
- изображения корзин с яблоками, преобразованные в изображения корзин с апельсинами и наоборот;

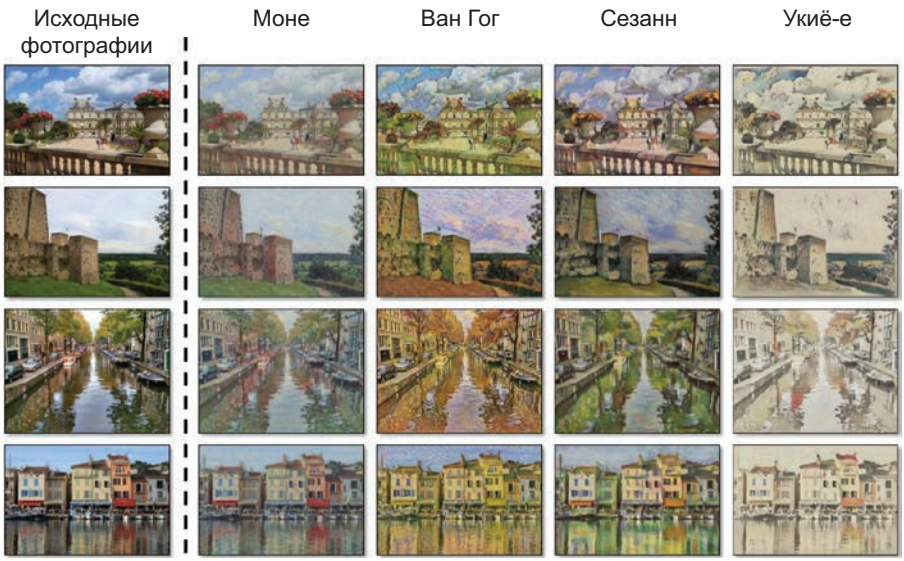


Рис. 3.5. Фотографии, преобразованные сетью CycleGAN в картины в стиле известных художников

[1] Сеть получила название «CycleGAN», потому что сохраняет целостность изображения в течение множества циклов обучения. *Zhu J.-Y.* et al. (2017). «Unpaired image-to-image translation using cycle-consistent adversarial networks». *arXiv:1703.10593*.

- плоские фотографии низкого качества, преобразованные в фотографии, снятые профессиональными (однообъективными зеркальными) камерами;
- видео лошади, бегущей по полю и постепенно превращающейся в зебру;
- дневную видеозапись, преобразованную в ночную.

ПРИДАНИЕ ФОТОРЕАЛИСТИЧНОСТИ ПРОСТЫМ РИСУНКАМ

Еще одним забавным примером применения GAN может служить приложение *pix2pix*[1], разработанное в лаборатории BAIR Алексеем Эфросом. Перейдя по ссылке bit.ly/pix2pixDemo, вы сможете в интерактивном режиме преобразовать простые рисунки в фотореалистичные изображения и обратно. Например, используя инструмент edge2cats, мы нарисовали трехглазого кота (слева на рис. 3.6) и сгенерировали фотореалистичное изображение (справа). При желании вы тоже можете попробовать преобразовать свои рисунки кошек, обуви, сумок и фасадов зданий в фотореалистичные аналоги прямо в браузере. Авторы статьи с описанием pix2pix назвали свой подход *условная* GAN (conditional GAN или просто cGAN), потому что генеративно-состязательная сеть производит результат, зависящий от конкретного входного изображения.

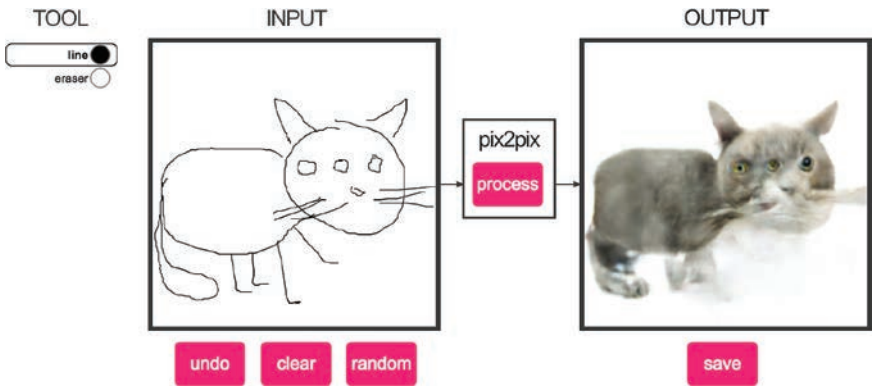


Рис. 3.6. Трехглазый кот-мутант (справа), синтезированный веб-приложением pix2pix. Рисунок слева, который был передан в сеть GAN, нарисован не иллюстратором этой книги Аглаэ, а одним из остальных авторов (не будем называть его)

[1] *Isola P.* et al. (2017). «Image-to-image translation with conditional adversarial networks». *arXiv:1611.07004*.

СОЗДАНИЕ ФОТОРЕАЛИСТИЧНЫХ ИЗОБРАЖЕНИЙ ИЗ ТЕКСТА

В завершение этой главы мы предлагаем взглянуть на действительно фотореалистичные изображения на рис. 3.7. Они были сгенерированы сетью StackGAN[1], *состоящей из двух сетей* GAN, расположенных друг над другом. Архитектура первой сети настроена для создания грубого изображения с низким разрешением, передающего общую форму и цвета соответствующих объектов. Это изображение затем передается второй сети, которая уточняет фиктивную «фотографию», устраняя недостатки и добавляя детали. StackGAN — это cGAN, подобная сети pix2pix, описанной в предыдущем разделе; но она генерирует изображения на основе текстовых описаний, а не изображений.

Рис. 3.7. Фотореалистичные изображения с высоким разрешением, полученные сетью StackGAN, которая состоит из двух расположенных друг над другом сетей GAN

ОБРАБОТКА ИЗОБРАЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИЙ ГЛУБОКОГО ОБУЧЕНИЯ

Со времени появления цифровых камер обработка изображений (как во время съемки, так и после) стала одним из основных в большинстве (если не во всех) рабочих процессов фотографов. К их числу относятся и простое применение

[1] *Zhang H.* et al. (2017). «StackGAN: Text to photo-realistic image synthesis with stacked generative adversarial networks». *arXiv:1612.03242v2*.

эффектов во время съемки, например увеличение насыщенности и резкости сразу после захвата, и сложное редактирование исходных файлов с изображениями в таких программных приложениях, как Adobe Photoshop и Lightroom.

Машинное обучение широко применяется для обработки во время съемки, потому что производители камер стремятся обеспечить высокое качество снимков при минимальных усилиях пользователя. Вот некоторые примеры:

- алгоритмы раннего распознавания лиц в фотокамерах типа «наведи и снимай», которые оптимизируют экспозицию и резкость по лицам и даже иногда спускают затвор, когда распознают улыбки (как на рис. 1.13);
- алгоритмы оценки сцены, которые корректируют настройки экспозиции, чтобы точнее отразить белизну снега, или активируют вспышку при съемке в темноте.

В настоящее время существует множество автоматизированных инструментов обработки фотографий после съемки, но при этом фотографам, которые уделяют внимание постобработке, приходится вкладывать время и знания предметной области для коррекции цвета и экспозиции, удаления шума и повышения резкости (и это лишь некоторые из применяемых методов).

Исторически эти корректировки было трудно выполнить программно, потому что шумоподавление, например, часто требуется выборочно применять к разным изображениям и даже к разным частям одного изображения. Именно с этой задачей способно справиться интеллектуальное приложение, использующее технологии глубокого обучения.

В своей статье, опубликованной в 2018 году, Чен Чен (Chen Chen) и его коллеги из Intel Labs[1] рассказали о применении глубокого обучения для улучшения изображений, полученных почти в полной темноте, с удивительными результатами (рис. 3.8). Их модель глубокого обучения включает сверточные слои, организованные в инновационную архитектуру *U-Net*[2] (которую мы исследуем в главе 10). Авторы создали специальный набор данных для обучения модели, включающий 5094 необработанные фотографии очень темных сцен, полученных с короткой[3] и большой выдержкой (для стабилизации использовался штатив). Время экспозиции при съемке с большой выдержкой в 100–300 раз превышало время экспозиции при фотографировании с короткой выдержкой и фактически составляло от 10 до 30 секунд. Как показано на рис. 3.8, конвейер обработки изображений U-Net на основе глубокого обучения (справа) значительно пре-

---

[1] *Chen C.* et al. (2018) «Learning to see in the dark». *arXiv:1805.01934*.

[2] Ronneberger et al. (2015) «U-Net: Convolutional networks for biomedical image segmentation». *arXiv:1505.04597*.

[3] То есть достаточно короткой, чтобы съемку можно было произвести с рук без размытия, но из-за этого изображения получаются слишком темными, чтобы на них можно было что-то разглядеть.

взошел результаты, полученные традиционным конвейером (в центре). Однако есть ограничения:

- модель недостаточно быстра и не способна выполнять корректировки в режиме реального времени (и уж точно неспособна выполнять их во время съемки на устройстве); однако здесь может помочь оптимизация времени выполнения;
- сеть требует обучения для каждой конкретной камеры, тогда как предпочтительнее было бы иметь более обобщенное решение, не зависящее от модели;
- несмотря на существенное превосходство перед традиционными конвейерами, на исправленных фотографиях все еще присутствуют некоторые искажения, которые можно было бы устранить;
- набор данных ограничен выбранными статическими сценами, и его желательно дополнить другими объектами (в частности, изображениями людей).

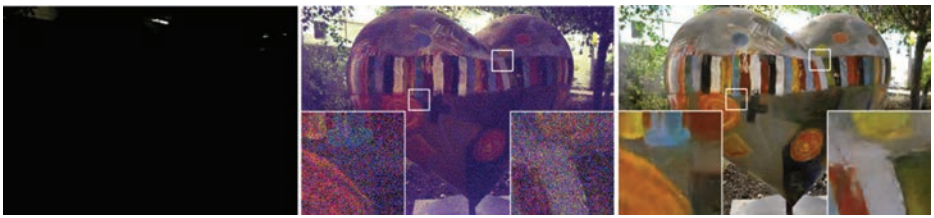


Рис. 3.8. Исходное изображение (слева), обработанное с использованием традиционного конвейера (в центре) и модели глубокого обучения, созданной Ченом с коллегами (справа)

Однако, несмотря на ограничения, эта работа позволяет с оптимизмом рассматривать возможность применения глубокого обучения для адаптивной корректировки изображений в конвейерах постобработки фотографий с уровнем сложности, ранее недостижимом для машин.

ИТОГИ

В этой главе мы познакомились с генеративно-состязательными сетями (GAN) и выяснили, что этот метод глубокого обучения основан на создании исключительно сложных представлений в скрытых пространствах. Эти богатые визуальные представления позволяют сети GAN создавать новые изображения с особыми художественными стилями. Результаты, полученные сетью GAN, имеют не только эстетический, но и практический характер. Они могут, например, моделировать данные для обучения беспилотных транспортных средств,

ускорять создание прототипов в области моды и архитектуры и существенно увеличивать творческие возможности людей[1].

В главе 12, после знакомства с необходимой теорией глубокого обучения, вы сами создадите генеративно-состязательную сеть, имитирующую рисунки из набора данных Quick, Draw! (о котором рассказывалось в конце главы 1). На рис. 3.9 показаны примеры того, что вы сможете сделать.

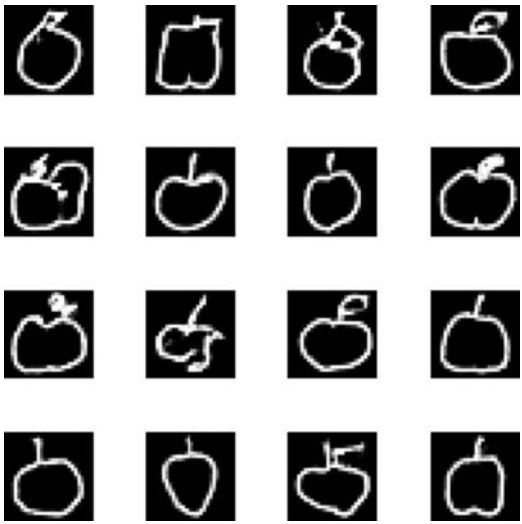


Рис. 3.9. Имитации рисунков яблок, созданные с помощью архитектуры GAN, которую мы разработаем вместе с вами в главе 12. Используя этот подход, вы сможете обучить машину создавать «рисунки» в любой из сотен категорий, присутствующих в игре Quick, Draw!

[1] *Carter S. and Nielsen M.* (2017, December 4). «Using artificial intelligence to augment human intelligence». Distill. distill.pub/2017/aia

4

МАШИНЫ-ИГРОКИ

Достижению потрясающих успехов в области искусственных нейронных сетей, включая прорывы в области искусственного интеллекта, совершенные в последние годы (наряду с генеративно-состязательными сетями, представленными в главе 3), способствовало глубокое *обучение с подкреплением*. В этой главе мы расскажем, что такое обучение с подкреплением и как его использование в сочетании с глубоким обучением позволило машинам достичь уровня человека и даже превзойти его в решении самых сложных задач, включая видеоигры для Atari, настольную игру Go и выполнение точнейших манипуляций.

ГЛУБОКОЕ ОБУЧЕНИЕ, ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ И ДРУГИЕ

Выше мы познакомились с примерами применения глубокого обучения в сфере компьютерного зрения (глава 1), обработки естественного языка (глава 2) и художественного творчества (глава 3). При этом мы почти не упоминали применение глубокого обучения для реализации искусственного интеллекта. Теперь, приступая к обсуждению глубокого обучения с подкреплением, мы должны дать более точное определение этим терминам, а также подчеркнуть отношения между ними. Как обычно, нам будут помогать визуальные подсказки — в данном случае диаграмма Венна на рис. 4.1.

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Искусственный интеллект (*ИИ*) — это самый разрекламированный, неопределенный и широкий термин из всех, что мы рассмотрим в этом разделе. Независимо от технического определения, об ИИ можно с уверенностью сказать, что эта технология предполагает машинную обработку информации из окружающей среды и последующее ее использование для достижения желаемого результата.

Рис. 4.1. Диаграмма Венна, показывающая относительное расположение основных понятий, рассматриваемых в этой книге

Возможно, из-за этого неформального определения некоторые считают, что целью ИИ является достижение уровня «общего интеллекта» — интеллекта, под которым понимается наличие широких способностей к рассуждению и решению задач[1]. На практике, особенно в популярных СМИ, приставка «ИИ» используется для описания любых передовых возможностей машины. В настоящее время к таким возможностям относят распознавание голоса, описание происходящего в видеофильмах, ответы на вопросы, управление автомобилем, промышленных роботов, подражающих действиям человека, доминирование над людьми в «интеллектуальных» настольных играх, таких как Go. Когда эти возможности становятся обычным явлением (как, например, распознавание рукописных цифр, считавшееся прорывным достижением в 1990-х годах; см.

[1] Определение понятия «интеллект» — сложная задача, и обсуждение разных интерпретаций этого понятия выходит за рамки этой книги. Однако есть давнее определение термина, которое мы считаем забавным и которое до сих пор имеет сторонников в среде экспертов: «интеллект — это то, что измеряется тестами IQ». См., например: *van der Mass H.* et al. (2014). «Intelligence is what the intelligence test measures. Seriously» Journal of Intelligence, 2, 12–15.

главу 1), пресса перестает использовать приставку «ИИ» при их упоминании, из-за чего акценты в определении ИИ постоянно смещаются.

МАШИННОЕ ОБУЧЕНИЕ

Машинное обучение — один из разделов ИИ, так же как робототехника. Машинное обучение — это область информатики, связанная с созданием и настройкой программного обеспечения для распознавания шаблонов в данных, при этом программисту не нужно явно указывать, как программное обеспечение будет обрабатывать все аспекты, касающиеся распознавания. Однако обычно программист должен иметь некоторое представление о способах решения проблемы и представить грубую модель и соответствующие данные, чтобы обучающееся программное обеспечение было хорошо подготовлено к решению проблемы. Как показано на рис. 1.12 и не раз обсуждалось в предыдущих главах, машинное обучение традиционно включает ручную кропотливую и трудоемкую обработку исходных данных для извлечения признаков, которые хорошо сочетаются с алгоритмами моделирования данных.

ОБУЧЕНИЕ ПРЕДСТАВЛЕНИЮ

Сняв следующий слой с «луковицы» на рис. 4.1, обнаруживаем *обучение представлению*. Этот термин был введен в начале главы 2, поэтому здесь мы не будем вдаваться в подробности. Но напомню, что обучение представлению — это ветвь машинного обучения, в которой модели строятся так, что при наличии достаточного объема данных они автоматически изучают признаки (или *представления*). Эти признаки могут оказаться как более тонкими, так и более обобщенными, чем их родственники, определяемые вручную. Признаки, изученные автоматически, часто менее понятны и сложнее поддаются объяснению, хотя исследователи из академических и промышленных кругов все чаще успешно справляются с этими трудностями[1].

ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

В настоящее время *искусственные нейронные сети* (Artificial Neural Network, ANN) доминируют в области обучения представлению. Как отмечалось в предыдущих главах и более подробно будет описано в главе 6, искусственные нейроны — это простые алгоритмы, имитирующие биологические клетки мозга, в том смысле что отдельные нейроны (биологические или искусственные) получают данные от многих других, выполняют некоторые вычисления и выводят единственный сигнал. Соответственно, искусственная нейронная сеть — это совокупность ис-

[1] См., например: *Kindermans P.-J.* et al. (2018). «Learning how to explain neural networks: PatternNet and PatternAttribution». International Conference on Learning Representations.

кусственных нейронов, организованных так, что они обмениваются информацией друг с другом. Исходные данные (например, изображения рукописных цифр) подаются в искусственную нейронную сеть, которая каким-то образом обрабатывает их с целью получения некоторого желаемого результата (например, для прогнозирования цифры, представленной на изображении).

ГЛУБОКОЕ ОБУЧЕНИЕ

Из всех терминов на рис. 4.1 *глубокое обучение* определить проще всего, потому что этот термин самый точный. Выше уже упоминалось, что сеть, состоящую хотя бы из нескольких слоев искусственных нейронов, уже можно назвать сетью глубокого обучения. Как показано на примере классических архитектур, на рис. 1.11 и 1.17, схематически подчеркнуто на рис. 4.2 и полностью будет описано в главе 7, сети глубокого обучения имеют в общей сложности пять или более уровней со следующей структурой:

- Один *входной* слой, который зарезервирован для передачи данных в сеть.
- Три или более *скрытых* слоев, которые обучаются представлениям во входных данных. Часто в роли скрытого слоя используется универсальный *плотный* (dense) слой, все нейроны которого могут получать информацию от каждого из нейронов в предыдущем (по этой причине для обозначения плотных слоев часто используется термин *полносвязанный слой*). Кроме этого универсального типа скрытых слоев существует масса специализированных типов для особых случаев использования; мы перечислим наиболее популярные из них далее в этом разделе.
- Один *выходной* слой, который зарезервирован для вывода значений (например, прогнозов) из сети.

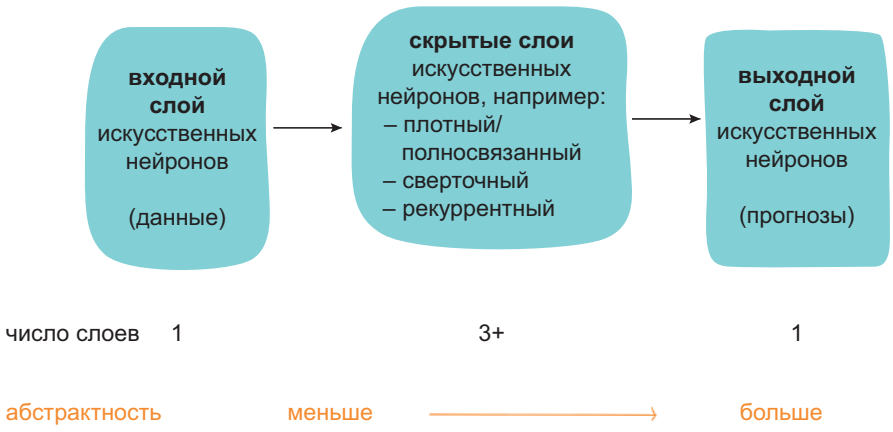


Рис. 4.2. Обобщенная архитектура моделей глубокого обучения

Каждый последующий слой в сети может представлять все более абстрактные, нелинейные рекомбинации предыдущих слоев, поэтому для изучения представлений, имеющих ценность для решаемой задачи, часто достаточно моделей глубокого обучения, имеющих меньше десятка слоев искусственных нейронов. Однако на практике было доказано, что в некоторых случаях имеет смысл конструировать сети глубокого обучения с сотнями или даже тысячами слоев[1].

Как показал стремительный рост точности и бесчисленные победы в состязаниях после победы AlexNet в ILSVRC в 2012 году (рис. 1.15), подход к моделированию на основе глубокого обучения прекрасно справляется с решением широкого спектра задач машинного обучения. Действительно, благодаря глубокому обучению, которое в значительной степени способствует прогрессу возможностей ИИ, в масс-медиа названия «глубокое обучение» и «искусственный интеллект» используются практически как взаимозаменяемые.

Давайте зайдем внутрь кольца «Глубокое обучение» на рис. 4.1 и рассмотрим классы задач, для которых используются алгоритмы глубокого обучения: компьютерное зрение, обработка естественного языка и обучение с подкреплением.

КОМПЬЮТЕРНОЕ ЗРЕНИЕ

Компьютерное зрение, реализованное по аналогии с системой биологического зрения, было представлено в главе 1. Основное внимание в этой главе уделялось задачам распознавания объектов, таких как рукописные цифры или породы собак.

Другими яркими примерами практического применения алгоритмов компьютерного зрения могут служить автомобили с автоматическим управлением, подсказки на основе распознавания объектов в фотоаппаратах и разблокировка смартфона с помощью распознавания лица его владельца. Вообще говоря, компьютерное зрение так или иначе связано с любым ИИ, которому требуется распознавать объекты по их внешнему виду или ориентироваться в реальной среде.

Сверточные нейронные сети (Convolutional neural network, ConvNet или CNN) являются важной разновидностью архитектур глубокого обучения для современных приложений компьютерного зрения. CNN — это модель глубокого обучения с любой архитектурой, которая имеет скрытые слои *сверточного* типа. Мы упомянули сверточные слои, когда обсуждали результаты, полученные генеративно-состязательной сетью Яна Гудфеллоу и представленные на рис. 3.2, и более подробно обсудим их в главе 10.

---

[1] См., например: *He K.* et al. (2016). «Identity mappings in deep residual networks». *arXiv:1603.05027*.

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

Обработку естественного языка (NLP) мы рассмотрели в главе 2. Глубокое обучение не имеет такого же однозначного преимущества в обработке естественного языка, как в компьютерном зрении, поэтому на диаграмме Венна (рис. 4.1) эта сфера изображена на пересечении глубокого обучения и более обширной области машинного обучения. Однако, как показано на хронологической шкале на рис. 2.3, подходы к NLP на основе глубокого обучения начинают опережать традиционное машинное обучение и с точки зрения эффективности, и с точки зрения точности. Фактически в таких областях обработки естественного языка, как распознавание речи (например, в Amazon Alexa или Google Assistant), машинный перевод (включая голосовой перевод в реальном времени по телефону) и поиск в Интернете (например, предсказание символов или слов, которые будут набраны пользователем), глубокое обучение уже заняло лидирующие позиции. В общем случае глубокое обучение в NLP имеет прямое отношение к любому ИИ, который поддерживает взаимодействия на естественном языке — устном или печатном, — в том числе и для автоматического поиска ответов на сложные вопросы.

В качестве скрытых слоев во многих архитектурах глубокого обучения в сфере NLP используются ячейки *с долгой краткосрочной памятью* (Long Short-Term Memory, LSTM), относящиеся к семейству *рекуррентных нейронных сетей* (Recurrent Neural Network, RNN). RNN можно применять для обработки любых данных, имеющих последовательную форму, как, например, финансовые данные за некоторый период времени, уровни запасов, плотность движения и погода. Более подробно мы рассмотрим RNN, в том числе LSTM, в главе 11, когда добавим их в прогнозирующие модели, обрабатывающие данные на естественном языке. Эти примеры послужат вам прочной базой, даже если вы собираетесь применять методы глубокого обучения к другим классам последовательных данных.

ТРИ КАТЕГОРИИ ЗАДАЧ МАШИННОГО ОБУЧЕНИЯ

На диаграмме Венна, рис. 4.1, остался еще один раздел, включающий обучение с подкреплением, которое является основной темой остальной части этой главы. Для знакомства с этим понятием мы сравним его с двумя другими основными категориями задач, для решения которых часто используются алгоритмы машинного обучения: обучение с учителем и без учителя.

ОБУЧЕНИЕ С УЧИТЕЛЕМ

В задачах *обучения с учителем* есть две переменные, x и y, где:

- x представляет данные, подаваемые на вход модели;
- y представляет результат, который должна вернуть модель; по этой причине переменную y еще называют *меткой*.

Цель обучения с учителем состоит в том, чтобы обучить модель аппроксимировать y по x. Задачи обучения с учителем обычно делят на два типа:

- *Регрессия*, когда переменная y изменяется в непрерывном диапазоне. Примерами могут служить прогноз объема продаж, будущей цены актива, такого как дом (этот пример приводится в главе 9), или доли в компании, котирующейся на бирже.
- *Классификация*, когда переменная y принимает дискретные значения меток, относящих каждый экземпляр x к определенной категории. Другими словами, y — это так называемая *категориальная переменная*. Примерами может служить идентификация рукописных цифр (эту модель мы создадим в главе 10) или прогноз о том, насколько понравится тот или иной фильм определенному человеку (будет показано в главе 11).

ОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

Задачи *обучения без учителя* отличаются отсутствием метки y. То есть в задачах обучения без учителя имеются некоторые данные x, которые можно передать в модель, но нет результата y, который нужно предсказать. Цель обучения без учителя состоит в том, чтобы обучить модель обнаруживать некоторую скрытую базовую структуру в данных. Типичным примером является группировка новостных статей по темам. Вместо подготовки предопределенного списка категорий, к которым относятся новостные статьи (политика, спорт, финансы и т. д.), модель настраивается для автоматической группировки по похожим темам. Другими примерами могут служить создание векторного пространства слов (см. главу 2) из данных на естественном языке (мы сделаем это в главе 11) или новых изображений с использованием генеративно-состязательной сети (как в главе 12).

ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ

Теперь мы готовы приступить к обсуждению задач *обучения с подкреплением*, которые заметно отличаются от перечисленных выше. Как показано на рис. 4.3, к сфере обучения с подкреплением можно отнести такие задачи, в которых требуется, чтобы *агент* выполнял последовательность действий в некотором *окружении*. Агент может, например, быть человеком или алгоритмом, играющим в видеоигру для Atari или управляющим автомобилем. Пожалуй, главное отличие обучения с подкреплением от обучения с учителем и без учителя состоит в том, что действия, предпринимаемые агентом, влияют на информацию, которую окружение передает агенту, то есть агент получает прямую обратную связь с результатами своих действий. В обучении с учителем и без учителя, напротив, модель никогда не влияет на исходные данные; она просто потребляет их.

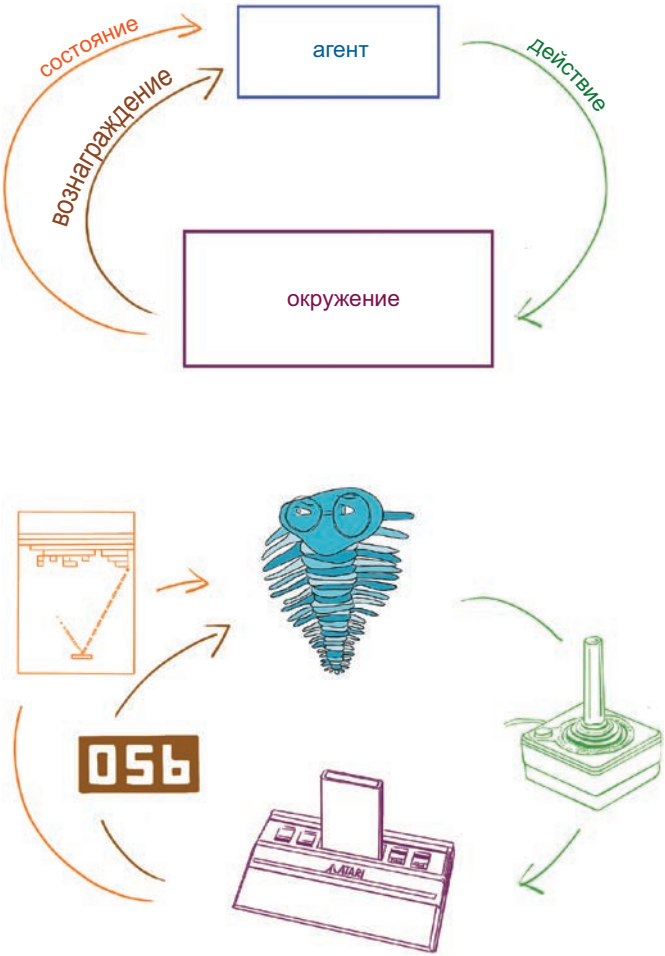


Рис. 4.3. Цикл обучения с подкреплением. Верхняя диаграмма — это обобщенная версия. Нижняя диаграмма иллюстрирует упомянутый в тексте пример агента, играющего в видеоигру на консоли Atari. Насколько нам известно, трилобиты не могут играть в видеоигры; мы используем трилобита лишь в качестве символического представления агента в обучении с подкреплением, которым может быть человек или машина

У студентов, изучающих теорию и практику глубокого обучения, часто возникает инстинктивное желание разделить парадигмы обучения с учителем, без учителя и с подкреплением на подходы, как основанные на традиционном и глубоком машинном обучении. В частности, они обычно связывают обучение с учителем с традиционным машинным обучением, а обучение без учителя или обучение с подкреплением (или оба) — с глубоким обучением. Сразу отмечу, что такой прямой связи между парадигмами не существует. Методы как традиционного машинного, так и глубокого обучения с успехом могут применяться к задачам обучения с учителем, без учителя и с подкреплением.

Давайте еще немного углубимся в отношения между агентом в обучении с подкреплением и его окружением, изучив некоторые примеры. На рис. 4.3 агент представлен антропоморфизированным трилобитом, но он может быть человеком или машиной. Этот агент играет в видеоигру для Atari:

- К возможным *действиям* относятся нажатия кнопок на игровом пульте[1].
- *Окружение* (консоль Atari) возвращает информацию агенту. Эта информация имеет две разновидности: *состояние* (пикселы на экране, представляющие текущее состояние окружения) и *вознаграждение* (количество очков в игре, которое агент стремится максимизировать с помощью игрового процесса).
- Если предположить, что агент играет в игру Pac-Man, тогда выбор действия, соответствующего нажатию кнопки «вверх», приводит к тому, что окружение возвращает обновленное состояние, в котором пикселы, представляющие персонажа видеоигры на экране, переместились вверх. До начала игры типичный алгоритм обучения с подкреплением даже не знает об этой простой взаимосвязи между кнопкой «вверх» и движением вверх персонажа в игре Pac-Man; обучение протекает с нуля, методом проб и ошибок.
- Если агент выберет действие, которое заставит персонажа пересечься с парой вкуснейших вишен, окружение вернет *положительное вознаграждение*: увеличенное число очков. Напротив, если агент выберет действие, которое заставит персонажа пересечься с жутким призраком, окружение вернет *отрицательное вознаграждение*: уменьшенное число очков.

Во втором примере, где агент управляет автомобилем:

- Спектр доступных *действий* намного шире и богаче, чем в игре Pac-Man. Агент может изменять угол поворота руля, нажимать на педали акселератора и тормоза — иногда чуть-чуть, а иногда, что называется, до упора.
- *Окружением* в этом случае является реальный мир, в котором имеются дороги, другие автомобили, пешеходы, деревья, небо и т. д. Соответственно, состояние — это состояние среды, окружающей транспортное средство, воспринимаемое глазами и ушами человека-агента или камерами и лидаром[2] автопилота.
- В случае алгоритма можно запрограммировать *положительное вознаграждение*, например, за каждый метр расстояния, пройденного в направлении

---

[1] Нам не известны алгоритмы игры в видеоигры, которые буквально нажимали бы на кнопки на пульте игровой приставки. Обычно они взаимодействуют с видеоигрой напрямую, используя программную эмуляцию. В конце главы мы перечислим самые популярные пакеты с открытым исходным кодом, предназначенные для этого.

[2] Лазерный радар.

точки назначения; оно может иметь небольшую *отрицательную* величину за незначительные нарушения правил дорожного движения и существенно отрицательную в случае столкновения.

ГЛУБОКОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ

Наконец мы достигли *глубокого обучения с подкреплением*, находящегося вблизи центра на диаграмме Венна на рис. 4.3. Алгоритм обучения с подкреплением получает приставку «глубокий», когда в своей работе использует искусственную нейронную сеть, например, чтобы узнать, какие действия нужно предпринять при том или ином состоянии окружения для увеличения вероятности получить положительное вознаграждение[1]. Как вы увидите в примерах, представленных в следующем разделе, попытка объединить подходы глубокого обучения и обучения с подкреплением оказалась весьма успешной, потому что:

- Глубокие нейронные сети превосходно справляются с обработкой сложных сенсорных входных данных, предоставляемых реальными или достаточно совершенными моделируемыми окружениями, выбирая релевантные сигналы из какофонии входных данных. Они подобны биологическим нейронам зрительной и слуховой коры мозга, которые получают информацию от глаз и ушей соответственно.
- В то же время алгоритмы обучения с подкреплением успешно справляются с задачей выбора подходящего действия из широкого спектра возможностей.

Глубокое обучение и обучение с подкреплением вместе являются мощной комбинацией для решения задач. Как правило, чем сложнее задача, тем больше требуется данных, чтобы процесс глубокого обучения с подкреплением смог пробраться сквозь шум, искажения и высокую степень случайности, а в конце концов найти эффективную политику в выборе действий в определенных обстоятельствах. Поскольку многие задачи обучения с подкреплением решаются в моделируемом окружении, получение достаточного количества данных часто не является проблемой: агента можно просто обучить в дальнейших циклах моделирования.

Теоретические основы глубокого обучения с подкреплением были разработаны еще пару десятилетий назад[2], однако, как и в случае с AlexNet для глубокого

---

[1] Выше в этой главе (рис. 4.2) мы указывали, что термин «глубокое обучение» применяется к искусственным нейронным сетям, имеющим не менее трех скрытых слоев. Это условие обычно соблюдается в задачах обучения с подкреплением, тем не менее термин «глубокое обучение с подкреплением» можно использовать, даже если искусственная нейронная сеть, включенная в модель, имеет только один или два скрытых слоя.

[2] *Tesauro G.* (1995). «Temporal difference learning and TD-Gammon». Communications of the Association for Computing Machinery, 38, 58–68.

обучения (рис. 1.17), стремительное развитие глубокого обучения с подкреплением наблюдается лишь в последние несколько лет, что обусловлено стечением трех обстоятельств:

1. Экспоненциальный рост объемов наборов данных и появление более богатых моделируемых окружений.
2. Появление поддержки параллельных вычислений на графических процессорах (GPU) для эффективного моделирования с использованием больших наборов данных и широким выбором возможных состояний и действий.
3. Формирование исследовательской экосистемы, связавшей научные круги и промышленность и быстро воплощающей новые идеи как о глубоких нейронных сетях в целом, так и об алгоритмах глубокого обучения с подкреплением, в частности, например, для выбора оптимальных действий в широком спектре искаженных состояний.

ВИДЕОИГРЫ

Многие читатели этой книги наверняка могут вспомнить, как изучали новые видеоигры в детстве. Сидя в зале игровых автоматов или глядя на экран тяжеленного семейного телевизора с электронно-лучевой трубкой, вы быстро понимали, что пропуск мяча в игре Pong или Breakout ведет к проигрышу. Вы обрабатывали визуальную информацию на экране и стремились заработать дополнительные очки, чтобы превзойти достижения ваших друзей, и разрабатывали эффективные стратегии управления пультом для достижения этой цели. В последние годы исследователи из фирмы под названием DeepMind разрабатывают программное обеспечение, которое тоже учится играть в классические игры для Atari.

DeepMind — британский технологический стартап, основанный Демисом Хассабисом (Demis Hassabis, рис. 4.4), Шейном Леггом (Shane Legg) и Мустафой Сулейманом (Mustafa Suleyman) в 2010 году в Лондоне. Своей целью они поставили «решение загадки интеллекта», то есть были заинтересованы в расширении сферы ИИ путем создания все более универсальных алгоритмов обучения. Одной из их первых разработок стали сети глубокого *Q*-обучения (Deep Q-Learning, DQN; см. рис. 4.1). С их использованием модель с единой архитектурой научилась хорошо играть в несколько игр для Atari 2600 — с нуля, просто методом проб и ошибок.

В 2013 году Володимир Мних[1] (Volodymyr Mnih) и его коллеги из DeepMind опубликовали статью[2] об агенте DQN — подходе к глубокому обучению с под-

[1] Мних получил докторскую степень в Университете Торонто под руководством Джеффа Хинтона, рис. 1.16).

[2] *Mnih V.* et al. (2013). «Playing Atari with deep reinforcement learning». *arXiv: 1312.5602.*

креплением, с которым вы сможете подробно ознакомиться в главе 13, когда сами, шаг за шагом, создадите свой вариант агента. В качестве информации о *состоянии* их агент получал значения пикселов из *окружения*, эмулятора видеоигр[1], подобно человеку, играющему в Atari на экране телевизора. Для эффективной обработки этой информации в DQN была включена сверточная нейронная сеть (CNN) — типичный выбор для моделей глубокого обучения с подкреплением, получающих визуальную информацию (именно поэтому на рис. 4.1 мы изобразили области «Глубокое обучение с подкреплением» и «Компьютерное зрение» пересекающимися). Обработка потока визуальных данных из игр Atari (в данном случае чуть более двух миллионов пикселов в секунду) подчеркивает, насколько хорошо глубокое обучение подходит для выделения значимых признаков из шума. Кроме того, имитация игр Atari в эмуляторе — это задача, с которой, в частности, хорошо справляется глубокое обучение с подкреплением: несмотря на богатство возможных действий, спроектированных так, чтобы усложнить их освоение, к счастью, не существует конечного ограничения на объем доступных обучающих данных, потому что агент может продолжать играть до бесконечности.

Рис. 4.4. Демис Хассабис участвовал в основании DeepMind в 2010 году, после защиты кандидатской диссертации в области когнитивной неврологии в Университетском колледже Лондона

Во время обучения модель DeepMind DQN не получала никаких подсказок или стратегий — только состояние (пикселы изображения на экране), вознаграждение (число очков, для максимизации которого запрограммирована модель) и диапазон возможных действий (кнопки на пульте), доступных в данной игре Atari. Модель не настраивалась для конкретных игр, и все же ей удалось превзойти существующие подходы машинного обучения в шести играх из семи, на которых Мних и его коллеги проводили тестирование, и в трех играх даже превзойти опытных игроков-людей. Возможно, под влиянием такого заметного прогресса Google приобрела DeepMind в 2014 году за полмиллиарда долларов США.

В последующей статье, опубликованной в известном журнале *Nature*, Мних и его товарищи по команде в Google DeepMind оценили свой алгоритм DQN на 49 играх Atari[2]. Результаты показаны на рис. 4.5: алгоритм превзошел другие

[1] *Bellemare M.* et al. (2012). «The arcade learning environment: An evaluation platform for general agents». *arXiv: 1207.4708.*

[2] *Mnih V.* et al. (2015). «Human-level control through deep reinforcement learning». *Nature*, 518, 529–33.

подходы машинного обучения во всех играх, кроме трех (в 94% игр), и, что удивительно, на большинстве из них (59%) поднялся выше человеческого уровня.

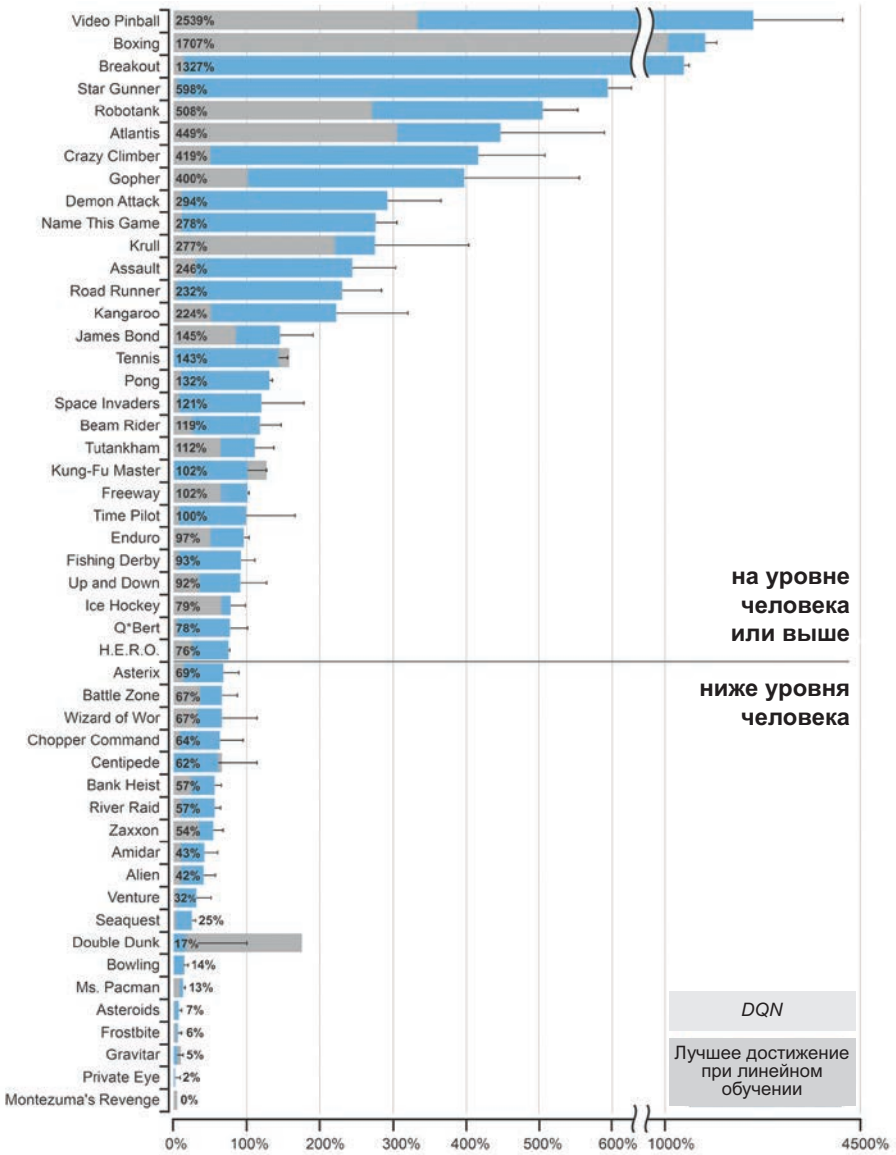


Рис. 4.5. Нормализованные показатели модели DQN, созданной Минхом и его коллегами (2015), относительно профессионального тестировщика игр: 0% представляет случайную игру, а 100% — лучшее достижение профессионала. Горизонтальная линия представляет «уровень человека», определенный автором: 75-й процентиль достижений профессионалов

НАСТОЛЬНЫЕ ИГРЫ

Считается, что настольные игры являются предтечей видеоигр, учитывая их аналоговую природу и давность появления; однако появление программных эмуляторов открыло простой и легкий путь к цифровому взаимодействию с видеоиграми. Доступность инструментов эмуляции обеспечила возможности, поэтому основные достижения в современном глубоком обучении с подкреплением первоначально появились в области видеоигр. Кроме того, сложность некоторых классических настольных игр, если сравнивать их с играми для Atari, намного выше. Существует множество стратегий и долгих сценариев для игр, напоминающих шахматы, которые внешне выглядят менее очевидными, чем Pac-Man или Space Invaders. В этом разделе мы посмотрим, как стратегии глубокого обучения с подкреплением способны овладевать такими настольными играми, как Го, шахматы и Сёги, несмотря на трудности с доступностью данных и сложностью вычислений.

ALPHAGO

Игра Го, изобретенная в Китае несколько тысячелетий тому назад, — это широко распространенная в Азии стратегическая настольная игра для двух игроков. Есть набор простых правил, основанных на идее захвата фигур противников (называются *камнями*), окружая их своими фигурами[1]. Однако на практике это несложное правило оказывается весьма противоречим и запутанным. Большая игровая доска и больший набор возможных ходов делают игру намного сложнее, чем, скажем, шахматы, для которых за два последних десятилетия были разработаны алгоритмы, способные победить лучших игроков-людей[2]. В Го имеется 2×10^{170} возможных позиций, что намного больше, чем число атомов во Вселенной[3] и примерно в *гугол* (10^{100}) раз больше, чем в шахматах.

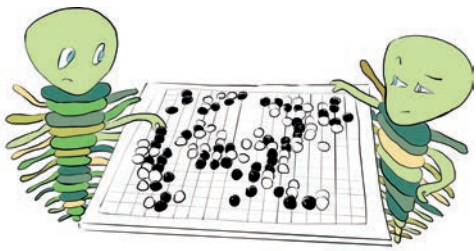


Рис. 4.6. Настольная игра Го. Один игрок играет белыми камнями, другой — черными. Цель состоит в том, чтобы окружить камни противника и захватить их

[1] В переводе с китайского «Го» буквально означает «настольная игра “окружение”».

[2] В 1997 году команда IBM Deep Blue одержала победу над Гарри Каспаровым, одним из величайших шахматистов в мире. Подробнее об этом легендарном матче будет рассказано чуть ниже в этом разделе.

[3] В наблюдаемой Вселенной насчитывается около 10^{80} атомов.

Сильвер с коллегами (2016) использовали обучение с учителем на архивных записях партий в Го, сыгранных людьми, чтобы создать так называемую *сеть стратегий* (policy network), которая генерирует короткий список возможных ходов в данной ситуации. Впоследствии сеть стратегий была усовершенствована применением методов глубокого обучения с подкреплением и проведением сеансов *игры с самой собой*, где оба соперника представлены агентами для игры в Го с сопоставимым уровнем квалификации. Играя с самим собой, агент постепенно совершенствуется, но, поднявшись на новый уровень, он вынужден противостоять более совершенному самому себе. В результате создается неразрывная петля положительной обратной связи. И наконец, вишенка на торте модели AlphaGo: так называемая *оценочная сеть* (value network), которая предсказывает победителя в играх с самой собой, оценивая позицию на доске и учась определять сильные ходы. Комбинация этих двух сетей (подробнее о них рассказывается в главе 13) сужает область поиска для MCTS.

Для поиска выигрышных ходов в несложных играх можно использовать алгоритм с названием *поиск Монте-Карло в дереве* (Monte Carlo Tree Search, MCTS). В чистом виде алгоритм MCTS предполагает выбор случайных ходов[1] до конца игрового процесса. В повторных сеансах игры ходам, которые обычно приводили к победе, присваивается больший вес. Из-за чрезвычайной сложности и огромного количества возможных вариантов в таких играх, как Го, использовать чистый алгоритм MCTS нецелесообразно: просто слишком много вариантов для поиска и оценки. Вместо чистого MCTS обычно используется альтернативное решение, включающее применение алгоритма MCTS к гораздо более узкому подмножеству действий, которое определяется, например, избранной стратегией оптимальной игры. Такого подхода оказалось достаточно для победы над игроками-любителями в Го, но он не может состязаться с профессионалами. Чтобы преодолеть разрыв между любителями и профессионалами, Дэвид Сильвер (David Silver, рис. 4.7) и его коллеги из Google DeepMind разработали программу AlphaGo, объединяющую MCTS и обучение с учителем и без учителя[2].

Модель AlphaGo сумела обыграть большинство других компьютерных программ для игры в Go. Но самое, пожалуй, поразительное, что AlphaGo смогла

Рис. 4.7. Дэвид Сильвер — научный сотрудник Google DeepMind, получивший образование в Кембридже и Альберте. Сыграл важную роль в объединении парадигм глубокого обучения и обучения с подкреплением

[1] Монте-Карло — район Монако, где находится множество казино. Это название напоминает о случайности результатов.

[2] *Silver D.* et al. (2016). «Mastering the game of Go with deep neural networks and tree search». Nature, 529, 484–9.

победить Фань Хуэя (Fan Hui), тогдашнего действующего чемпиона Европы по Го, со счетом 5:0. Это был первый случай, когда компьютер победил профессионального человека-игрока вчистую. Как показано на рис. 4.8, модель AlphaGo получила рейтинг Elo[1], который находится на уровне и даже выше уровня лучших игроков в мире.

Вслед за этим успехом в марте 2016-го в Сеуле (Южная Корея) модель AlphaGo вступила в схватку с Ли Седолом (Lee Sedol). Седол обладал 18 мировыми титулами и считался одним из величайших игроков всех времен. Матч из пяти партий транслировался в прямом эфире, и за ним наблюдали 200 миллионов

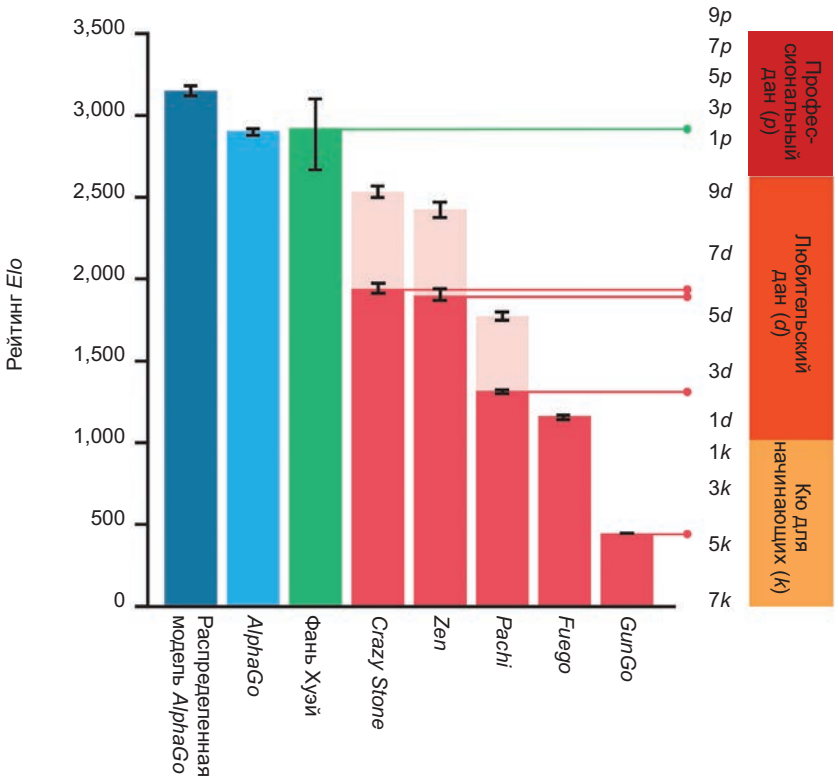


Рис. 4.8. Рейтинг Elo модели AlphaGo (синие столбики) и рейтинги Фань Хуэя (зеленый столбик) и некоторых других программ для игры в Го (красные столбики). Справа показана расшифровка рейтингов для людей

[1] Рейтинги Elo позволяют сравнивать уровень мастерства игроков-людей и алгоритмов. Они рассчитываются на основе личных побед и поражений. Человек с более высоким рейтингом имеет больше шансов выиграть у соперника с более низким рейтингом. Чем больше разрыв между игроками, тем выше вероятность, что игрок с более высоким рейтингом выйдет победителем.

человек. AlphaGo победила со счетом 4:1, обеспечив широкую известность DeepMind, Го и возможностям искусственного интеллекта в будущем[1].

ALPHAGO ZERO

После создания AlphaGo сотрудники DeepMind продолжили свою работу и разработали модель второго поколения для игры в Го: AlphaGo Zero. Напомню, что AlphaGo изначально создавалась с применением методов обучения с учителем; то есть первоначально модель обучалась на партиях, сыгранных профессиональными игроками-людьми, а затем использовались методы обучения с подкреплением путем проведения игр с самой собой. Несмотря на все изящество этого подхода, он не «решает загадку интеллекта», как хотелось бы основателям DeepMind. Лучшим приближением к универсальному интеллекту могла бы стать сеть, способная научиться играть в Го полностью *с нуля* — когда сеть не обучается на человеческом опыте или на знаниях предметной области, а совершенствуется только за счет глубокого обучения с подкреплением. В результате появилась модель AlphaGo Zero.

Как уже упоминалось выше, игра в Го требует серьезного умения прогнозировать ситуацию в обширном пространстве поиска. То есть *возможных* ходов так много, а *хороших* ходов так мало (и в короткой, и в долгой игре), что поиск оптимального хода с учетом вероятного развития игры становится чрезвычайно сложной и непрактичной в вычислительном отношении задачей. Именно по этой причине считалось, что Го станет последним рубежом для машинного интеллекта; на самом деле многие считали, что успех, достигнутый моделью AlphaGo в 2016 году, будет возможен не ранее чем через десять лет.

Вдохновленные победой в матче AlphaGo против Седола в Сеуле, исследователи из DeepMind создали модель AlphaGo Zero, которая, будучи революционной[2] во многих отношениях, обучилась игре в Го и намного превзошла уровень оригинальной модели AlphaGo. Прежде всего, она обучалась без привлечения любых данных из партий, сыгранных человеком, то есть она училась только методом проб и ошибок. Во-вторых, в качестве входных данных использовалось только расположение камней на доске. В отличие от нее, модель AlphaGo получала 15 дополнительных признаков, спроектированных человеком, которые обеспечивали ключевые подсказки алгоритму, такие как количество ходов для прогнозирования вперед или количество камней противника, которое можно захватить. В-третьих, для оценки позиции и принятия решения о следующем ходе в AlphaGo Zero используется единственная (глубокая) нейронная сеть, а не

---

[1] Об этом матче с Седолом снят замечательный документальный фильм, который заставляет задуматься о многом. Режиссер Грег Кос (Kohs G.). «AlphaGo». Производство: США, Moxie Pictures & Reel As Dirt (2017).

[2] *Silver D.* et al. (2016). «Mastering the game of Go without human knowledge». Nature 550, 354–359.

отдельные сети стратегий и оценки (о которых упоминалось во врезке выше; подробнее о них рассказывается в главе 13). Наконец, поиск по дереву проще и опирается на нейронную сеть для оценки позиций и возможных ходов.

В течение трех дней AlphaGo Zero сыграла почти пять миллионов игр с самой собой, «обдумывая» каждый ход в течение примерно 0.4 секунды. За 36 часов она опередила модель, обыгравшую Ли Седола в Сеуле (задним числом названную AlphaGo Lee), которой в свое время потребовалось несколько месяцев для обучения. На 72-часовой отметке был проведен матч против AlphaGo Lee, в ходе которого она легко выиграла все 100 партий. Что самое примечательное, AlphaGo Zero достигла этого уровня совершенства, обучаясь на единственной машине, оснащенной четырьмя тензорными процессорами (TPU)[1], тогда как AlphaGo Lee обучалась в распределенной сети на нескольких машинах и использовала 48 TPU. (Модель AlphaGo Fan, опередившая Фань Хуэя, обучалась на 176 графических процессорах!) На рис. 4.9 показан рейтинг Elo для AlphaGo Zero по дням тренировок и сравнивается с результатами AlphaGo Master[2] и AlphaGo Lee. Справа показаны абсолютные рейтинги Elo для различных итераций AlphaGo и некоторых других программ для игры в Го. AlphaGo Zero — абсолютный лидер.

В результате этого исследования было сделано поразительное открытие: характер игры AlphaGo Zero качественно отличается от характера игры людей и AlphaGo Lee (обученной на человеческом опыте игры). Сначала AlphaGo Zero случайно выбирала ходы, но быстро освоила профессиональные *дзёсеки* — угловые последовательности, которые считаются признаком выдающейся игры. Однако после дальнейшего обучения зрелая модель стала отдавать предпочтение новым дзёсеки, которые ранее были неизвестны человечеству. Модель AlphaGo Zero сама выучила целый ряд классических ходов в Го, что означает высокую прагматичность этих приемов, но сделала это оригинальным способом: она освоила идею *шичо* (лестничные последовательности), например, намного позже, в то время как начинающих игроков-людей этому обучают в первую очередь. Авторы дополнительно обучили другую итерацию модели на данных об игре человека. Первоначально эта модель обучения с учителем показывала лучшие результаты, но затем, уже в течение первых 24 часов, стала уступать модели, обучавшейся без использования внешних данных, и в конечном итоге получила более низкий рейтинг Elo. Эти результаты показывают, что модель

[1] В Google создали свои процессоры для обучения нейронных сетей, известные как тензорные процессоры (TPU). Они взяли за основу архитектуру существующих GPU и оптимизировали ее для выполнения вычислений, преобладающих при обучении нейронных сетей. На момент написания TPU были доступны только через платформу Google Cloud.

[2] Модель AlphaGo Master является гибридом AlphaGo Lee и AlphaGo Zero; в ней используются дополнительные входные признаки, которыми пользуется AlphaGo Lee, и на начальном этапе она прошла обучение с учителем. В январе 2017 года AlphaGo Master приняла участие в анонимной онлайн-игре под псевдонимами Master и Magister и выиграла все 60 партий, играя против самых сильных игроков в Го.

с самообучением без внешних данных обретает стиль игры, отличный от стиля игроков-людей, — доминирующий стиль, который модель обучения с учителем не в состоянии постичь.

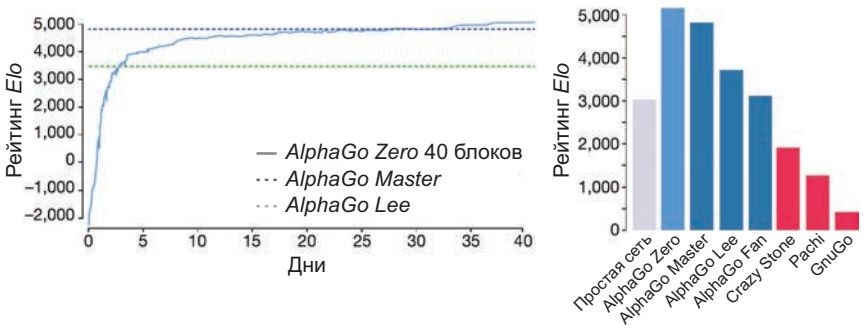


Рис. 4.9. Сравнение рейтинга Elo модели AlphaGo Zero с рейтингами других разновидностей AlphaGo и других программ для игры в Го. На графике слева сравнение проводится по дням обучения AlphaGo Zero

ALPHAZERO

Разгромив сообщество Го, команда DeepMind переключила свое внимание на универсальные игровые нейронные сети. После успеха, достигнутого моделью AlphaGo Zero в игре Го, они решили выяснить, можно ли научить сеть одинаково искусно играть в несколько игр. Для этого они добавили в свой репертуар две новые игры: шахматы и Сёги[1].

Многие читатели наверняка знакомы с игрой в шахматы; игра Сёги, которую еще называют «японскими шахматами», похожа на нее. Обе являются стратегическими играми для двоих, имеют игровое поле, разбитое на клетки, завершаются матом короля противника и включают ряд фигур с различными способностями к перемещению. Сёги, однако, значительно сложнее шахмат: она имеет игровое поле большего размера (9 × 9 против 8 × 8 в шахматах) и позволяет замещать фигуры противника в любом месте на доске после их захвата.

Исторически сложилось так, что искусственный интеллект тесно ассоциировался с игрой в шахматы. В течение нескольких десятилетий компьютерные программы для игры в шахматы получили широкое развитие. Самой известной из них является Deep Blue, созданная в IBM, которая в 1997 году одолела чемпиона мира Гарри Каспарова[2]. Она в значительной мере опиралась на

[1] *Silver D.* et al. (2017). «Mastering chess and shogi by self-play with a general reinforcement learning algorithm». *arXiv:1712.01815*.

[2] В 1996 году Deep Blue уступила Каспарову в первом матче, и только после внесения существенных усовершенствований одержала победу с небольшим отрывом. Это не было полной победой машины над человеком, на которое рассчитывали сторонники ИИ.

вычислительную мощь и метод прямого перебора[1], выполняя сложный поиск возможных ходов и комбинируя его с признаками, разработанными вручную, и некоторыми специфическими адаптациями. Deep Blue прошла тонкую настройку путем анализа тысяч игр гроссмейстеров (это была система обучения с учителем!) и подвергалась корректировке даже между играми[2].

Система Deep Blue была серьезным достижением два десятилетия тому назад, но не универсальным: она не могла решать никаких других задач, кроме игры в шахматы. После того как AlphaGo Zero продемонстрировала, что нейронная сеть может обучиться игре в Го сетью, опираясь только на теорию, не имея ничего, кроме игрового поля и правил игры, Сильвер и его коллеги из DeepMind решили разработать универсальную нейронную сеть, *единую архитектуру сети*, которая сможет побеждать не только в Го, но и в других настольных играх. По сравнению с игрой Го, шахматы и Сёги намного сложнее. Правила зависят от позиции (фигуры могут двигаться по-разному, в зависимости от своего местоположения на игровом поле) и сама игра асимметрична (некоторые фигуры могут двигаться только в одном направлении)[3]. Возможны атаки на большие расстояния (например, ферзь может пересекать всю доску), а игры могут заканчиваться ничьей.

AlphaZero вводила позиции фигур на доске в нейронную сеть и выводила вектор вероятностей ходов для каждого возможного действия, а также скалярную[4] оценку этого хода. Сеть изучала параметры этих вероятностей и оценок исключительно в ходе игр с самой собой методом глубокого обучения с подкреплением, как это делала AlphaGo Zero. Затем она выполняла поиск Монте-Карло (MCTS) в пространстве, сокращенном с учетом этих вероятностей, и возвращала уточненный вектор вероятностей возможных ходов. В отличие от AlphaGo Zero, которая оптимизировала вероятность выигрыша (Го — это игра с двумя исходами, выигрышем и проигрышем), модель AlphaZero оптимизировала конечный результат. Во время проведения игр с самой собой AlphaGo Zero сохраняла лучшего *игрока* за прошедший день и оценивала свои последующие обновленные версии, устраивая их поединки с этим игроком, постоянно заменяя игрока следующей лучшей версией. AlphaZero, напротив, поддерживала единую сеть и в каждый конкретный момент времени играла против последней версии самой себя. AlphaZero была обучена игре в шахматы, Сёги и Го всего за 24 часа. В нее не вносилось никаких корректировок, специфических для игры,

---

[1] На момент проведения матча с Каспаровым Deep Blue занимала 259-ю строчку в списке самых мощных суперкомпьютеров планеты.

[2] Эти корректировки стали предметом спора между IBM и Каспаровым после его проигрыша в 1997 году. В IBM отказались опубликовать журналы программы и демонтировали Deep Blue. Их компьютерная система никогда не получала официального рейтинга в шахматах, потому что сыграла слишком мало игр против шахматистов.

[3] Это усложняет расширение обучающих данных с использованием синтетического дополнения — подхода, широко использовавшегося в AlphaGo.

[4] Единственное значение.

кроме настраиваемого вручную параметра, определяющего, как часто модель должна выбирать случайные исследовательские ходы; это значение выражало отношение к количеству законных ходов в каждой игре[1].

Из 100 игр против Stockfish — лучшей компьютерной программы для игры в шахматы 2016 года, по версии Top Chess Engine Championship, — AlphaZero не проиграла ни одной. В матче Сёги чемпион мира по версии Computer Shogi Association, программа Elmo, смогла выиграть у AlphaZero всего восемь игр из ста. Самый, пожалуй, достойный противник, AlphaGo Zero, смогла победить AlphaZero в сорока из ста игр. На рис. 4.10 показаны рейтинги Elo для AlphaZero и этих трех ее соперниц.

AlphaZero оказалась не только самой результативной, но и самой эффективной. В рейтинге Elo модель AlphaZero поднялась выше наиболее сильных соперников после всего лишь двух, четырех и восьми часов обучения игре в Сёги, шахматы и Го соответственно. Это сенсационно высокая скорость обучения, учитывая, что компьютерные программы Elmo и Stockfish представляют собой кульминацию десятилетий исследований и тонкой настройки целенаправленным и специализированным для конкретной области образом. Универсальный алгоритм AlphaZero способен играть во все три игры и побеждать: простое переключение выученных весов в нейронных сетях с идентичными в остальном архитектурами наделяет каждую из них теми же навыками, на получение которых в других сетях ушли *годы*. Эти результаты демонстрируют поразительную мощь глубокого обучения с подкреплением для освоения игрового процесса на уровне эксперта нецеленаправленным способом.

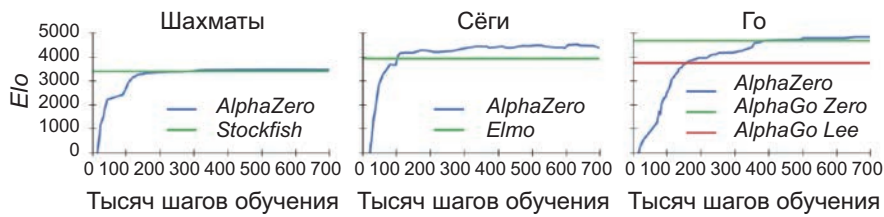


Рис. 4.10. Сравнение рейтингов Elo модели AlphaZero и всех ее соперников в шахматах, Сёги и Го. AlphaZero быстро опередила всех трех противников

МАНИПУЛИРОВАНИЕ ОБЪЕКТАМИ

До сих пор, в соответствии с названием этой главы, основное внимание мы уделяли знакомству с технологией обучения с подкреплением применительно

[1] Этот параметр, настраиваемый вручную, называется *epsilon*. Он подробно описывается в главе 13.

к играм. Безусловно, игры являются серьезным испытательным полигоном для исследования путей обобщения машинного интеллекта, но в этом разделе мы уделим немного времени знакомству с примерами практического применения глубокого обучения с подкреплением.

Выше в этой главе мы уже упомянули один реальный пример — транспортные средства с автопилотом. Дополнительно в этом разделе мы предоставим обзор исследований Сергея Левина (Sergey Levine), Челси Финн (Chelsea Finn, рис. 4.11) и сотрудников лаборатории Калифорнийского университета в Беркли[1]. Эти исследователи обучили робота выполнять множество движений, требующих визуального контроля и ощущения глубины, таких как навинчивание крышки на бутылку, извлечение гвоздя с помощью игрушечного молотка-гвоздодера, навешивание плечиков для одежды на перекладину и вставка объемных фигур в шаблон с подбором формы (рис. 4.12).

Алгоритм Левина, Финн и их коллег преобразует визуальное изображение непосредственно в сигналы, управляющие электромоторами в руке робота. Их сеть стратегий организована как семислойная сверточная нейронная сеть (CNN), состоящая из менее чем 100 000 искусственных нейронов, что очень немного по меркам глубокого обучения, как вы увидите далее в этой книге, когда будете обучать крупные сети на порядки больше этой. Было бы сложно подробно обсуждать этот подход до изучения теории искусственных нейронных сетей (во второй части книги, которая уже не за горами), поэтому мы не будем делать этого, но отметим три важных момента в этом элегантном практическом применении глубокого обучения с подкреплением. Во-первых, это «сквозная» модель глубокого обучения, в том смысле, что она получает необработанные изображения (пикселы) и формирует сигналы управления электромоторами робота. Во-вторых, модель достаточно точно обобщает широкий спектр уникальных задач манипулирования объектами. В-третьих, это пример семейства подходов *градиентного спуска по стратегиям* (policy gradient) к глубокому обучению с подкреплением, заключенных в одну область на диаграмме Венна на рис. 4.1. Методы градиентного спуска по стратегиям отличаются от глубокого *Q*-обучения (DQN), о котором рассказывается в главе 13, но мы также коснемся их.

Рис. 4.11. Челси Финн, аспирант Калифорнийского университета в Беркли, работает в исследовательской лаборатории ИИ

[1] *Levine S., Finn C.* et al. (2016). «End-to-end training of deep visuomotor policies». Journal of Machine Learning Research, 17, 1–40.

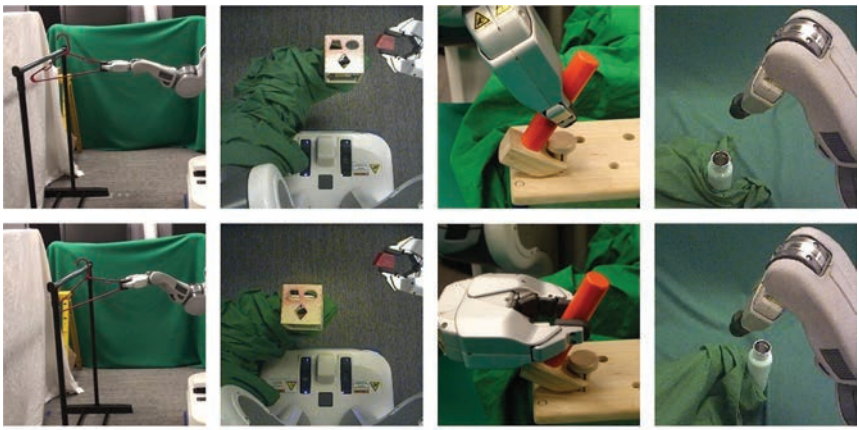


Рис. 4.12. Образцы изображений от Левина, Финн и др. (2016), демонстрирующие различные этапы манипулирования объектами, выполнению которых был обучен робот

ПОПУЛЯРНЫЕ ОКРУЖЕНИЯ ДЛЯ ГЛУБОКОГО ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ

В предыдущих нескольких разделах мы вскользь упомянули программную имитацию окружений, в которых протекает глубокое обучение моделей с подкреплением. Эта сфера имеет решающее значение для устойчивого развития обучения с подкреплением; без окружения, в котором агенты могут играть и заниматься исследованиями (и собирать данные!), модели не смогли бы обучаться. В этом разделе мы познакомимся с тремя наиболее популярными окружениями и обсудим их общие характеристики.

OPENAI GYM

Комплект OpenAI Gym[1] разработан некоммерческой исследовательской компанией OpenAI[2]. Задача OpenAI — продвижение универсального искусственного интеллекта (подробнее об этом в следующем разделе!) безопасным и беспристрастным способом. С этой целью исследователи в OpenAI разработали ряд инструментов с открытым исходным кодом для исследований в области ИИ, в том числе OpenAI Gym. Этот набор инструментов предоставляет интерфейс для обучения с подкреплением, как глубокого, так и обычного.

[1] github.com/openai/gym
[2] openai.com

Как показано на рис. 4.13, Gym включает разнообразные окружения, в том числе ряд игр для Atari 2600[1], несколько симуляторов роботов, несколько простых текстовых алгоритмических игр и несколько имитаций роботов, использующих физический движок MuJoCo[2]. В главе 13 мы установим OpenAI Gym и используем одно из его окружений для обучения нашего агента DQN. OpenAI Gym написан на Python и совместим со всеми библиотеками глубокого обучения, включая TensorFlow и PyTorch (библиотеки для глубокого обучения мы обсудим в главе 14; эти две пользуются особой популярностью).

DEEPMIND LAB

DeepMind Lab[3] — еще одно окружение для обучения с подкреплением от разработчиков из DeepMind (хотя они указывают, что DeepMind Lab не является официальным продуктом Google). Как показано на рис. 4.14, окружение построено на основе Quake III Arena[4] от id Software и имитирует научно-фантастический трехмерный мир, который могут исследовать агенты. Агент воспринимает окружение от первого лица, в отличие от эмуляторов Atari, доступных в OpenAI Gym.

В окружении имеется множество уровней, которые можно условно разделить на четыре категории:

1. Уровни сбора фруктов, на которых агент просто пытается найти и собрать награды (яблоки и дыни), стараясь избежать штрафов (лимоны).
2. Уровни навигации со статической картой, в которых агенту поручается найти цель и запомнить размещение объектов на карте. Перед началом эпизода агента можно поместить в произвольную точку на карте, а цель оставить на месте, чтобы проверить его способность повторно находить цель, опираясь только на память; или можно поместить агента в то же место, а цель перенести в другую позицию, чтобы проверить его способность к исследованиям.
3. Уровни навигации со случайными картами, в которых агент должен исследовать новую карту в каждом эпизоде, найти цель и затем повторно вернуться к цели столько раз, сколько возможно за определенное время.
4. Уровни лазерного боя, в которых агент вознаграждается за поиск и нападение на ботов в различных сценах.

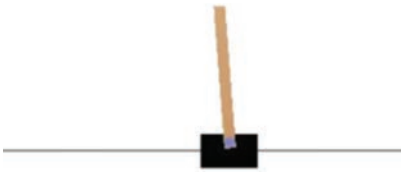
---

[1] Для эмуляции игр на Atari 2600 в OpenAI Gym используется Arcade Learning Environment. Этот же фреймворк используется в разработке Мниха с коллегами (2013), которая описана выше в разделе «Видеоигры». Этот фреймворк можно найти по адресу github.com/mgbellemare/Arcade-Learning-Environment.

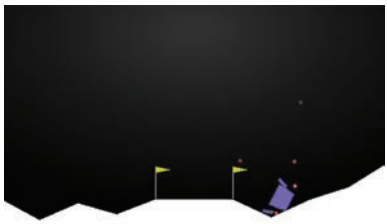
[2] MuJoCo, сокращенно от Multi-Joint dynamics with Contact, — это физический движок, разработанный Эмо Тодоровым (Emo Todorov) для Roboti LLC.

[3] *Beattie C.* et al. (2016). «DeepMind Lab». *arXiv:1612.03801.*

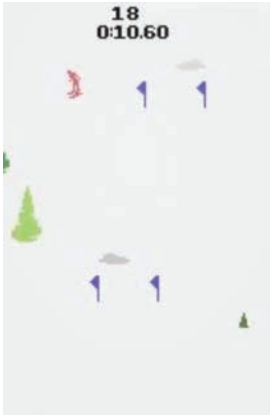
[4] Quake III Arena. (1999). United States: id Software. github.com/id-Software/Quake-III-Arena



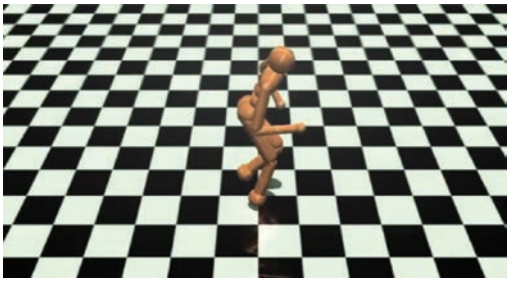
(a)



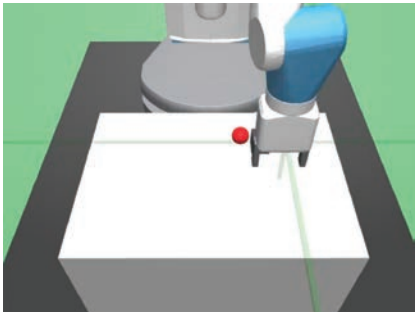
(b)



(c)



(d)



(e)

(f)

Рис. 4.13. Примеры из окружения OpenAI Gym: (a) CartPole, классическая задача из теории управления; (b) LunarLander, задача непрерывного управления в двумерном симуляторе; (c) Skiing, игра для Atari 2600, имитирующая спуск с горы на лыжах; (d) Humanoid, трехмерный симулятор механики двуногого человека на основе движка MuJuCo; (e) FetchPickAndPlace, одна из нескольких доступных имитаций роботов-манипуляторов в реальном мире, здесь показана имитация — Fetch, — задачей которой является захват блока и перемещение его в целевое местоположение; и (f) HandManipulateBlock, еще одна практическая имитация роботизированной руки, Shadow Dexterous Hand

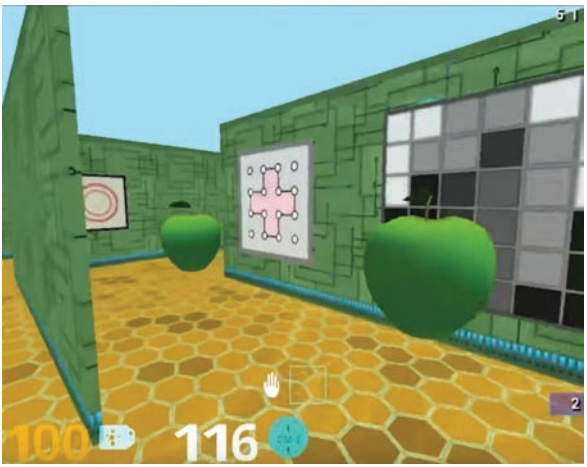


Рис. 4.14. Окружение DeepMind Lab, в котором очки начисляются за сбор зеленых яблок

Установить DeepMind Lab сложнее, чем OpenAI Gym[1], зато этот инструмент предоставляет богатое и динамичное окружение для обучения агентов от первого лица, а уровни предлагают сложные сценарии, требующие умения ориентироваться, запоминать, планировать стратегию и осуществлять мелкие движения. Эти сложные условия позволяют проверить границы возможностей современного глубокого обучения с подкреплением.

UNITY ML-AGENTS

Unity — мощный движок для двух- и трехмерных видеоигр и цифрового моделирования. Учитывая игровые навыки алгоритмов обучения с подкреплением, о которых рассказывалось выше в этой главе, неудивительно, что создатели популярного игрового движка тоже решили заняться созданием окружений для внедрения обучения с подкреплением в видеоигры. Плагин Unity ML-Agents[2] позволяет обучать модели в видеоиграх или симуляциях на основе Unity и, пожалуй, в большей степени соответствует целям самого Unity, позволяя моделям обучения с подкреплением управлять действиями агентов в игре. Подобно DeepMind Lab, плагин Unity ML-Agents устанавливается однострочной командой[3].

---

[1] Сначала нужно клонировать репозиторий Github (github.com/deepmind/lab), а затем скомпилировать программное обеспечение с использованием Bazel (bit.ly/installB). Подробные инструкции можно найти в репозитории DeepMind Lab (bit.ly/buildDML).

[2] github.com/Unity-Technologies/ml-agents

[3] Пользователь должен сначала установить Unity (инструкции по загрузке и установке можно найти по адресу store.unity.com/download), а затем клонировать репозиторий Github. Полные инструкции доступны в репозитории Github с исходным кодом Unity ML-Agents (bit.ly/MLagents).

ТРИ КАТЕГОРИИ ИИ

Глубокое обучение с подкреплением, пожалуй, наиболее тесно связано с популярными представлениями об искусственном интеллекте как о системе, обладающей когнитивными способностями человека к принятию решений. В свете этого и в завершение главы мы познакомимся с тремя категориями ИИ.

ОГРАНИЧЕННЫЙ ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Ограниченный искусственный интеллект (Artificial Narrow Intelligence, ANI) — это умение машин прекрасно справляться с конкретными задачами. В настоящее время существует множество разнообразных примеров ограниченного искусственного интеллекта, многие из которых уже упоминались в этой книге, такие как визуальное распознавание объектов, машинный перевод в режиме реального времени между естественными языками, автоматизированные торгово-финансовые системы, AlphaZero и автомобили с автоматическим управлением.

УНИВЕРСАЛЬНЫЙ ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Универсальный искусственный интеллект (Artificial General Intelligence, AGI) — это единый алгоритм, способный хорошо решать все задачи, описанные в предыдущем абзаце: он сможет распознать ваше лицо, перевести эту книгу на другой язык, оптимизировать ваш инвестиционный портфель, превзойти вас в игре в Го и безопасно довезти вас до места отдыха. Такой алгоритм действительно можно было бы сравнить с интеллектуальными способностями отдельного человека. Но нам придется преодолеть множество препятствий, чтобы создать универсальный искусственный интеллект, если его вообще удастся создать. Однако эксперты в области ИИ готовы тыкать пальцем в небо и давать прогнозы. В исследовании, проведенном философом Винсентом Мюллером (Vincent Müller) и влиятельным футуристом Ником Бостромом (Nick Bostrom)[1], большинство профессиональных исследователей ИИ сходятся в том, что универсальный искусственный интеллект будет создан к 2040 году.

ИСКУССТВЕННЫЙ СУПЕРИНТЕЛЛЕКТ

Искусственный суперинтеллект (Artificial Super Intelligence, ASI) трудно описать, потому что его трудно вообразить. Это мог бы быть алгоритм, значительно

[1] *Müller V. and Bostrom N.* (2014). «Future progress in artificial intelligence: A survey of expert opinion». In V. Müller (Ed.), Fundamental Issues of Artificial Intelligence. Berlin: Springer.

превосходящий интеллектуальные способности человека[1]. Если допустить, что универсальный искусственный интеллект возможен, тогда следует допустить и возможность искусственного суперинтеллекта. Конечно, на пути к суперинтеллекту еще больше препятствий, чем к универсальному ИИ, и основную их часть трудно предсказать. Тем не менее, ссылаясь на результаты тех же исследований Мюллера и Бострома, медианная оценка экспертов ИИ, когда появится искусственный суперинтеллект, находится около 2060 года — довольно гипотетическая дата, которая соответствует концу жизненного периода многих землян, живущих в настоящее время. В главе 14, когда вы уже хорошо будете разбираться в теории и практике глубокого обучения, мы обсудим вклад моделей глубокого обучения в создание универсального ИИ, а также существующие ограничения, связанные с глубоким обучением, которые необходимо преодолеть, чтобы создать универсальный искусственный интеллект или, чего уж там, искусственный суперинтеллект.

ИТОГИ

Эта глава началась с обзора связей между глубоким обучением и более широкой областью искусственного интеллекта. Затем мы поближе познакомились с глубоким обучением с подкреплением — подходом, сочетающим глубокое обучение с парадигмой обучения с подкреплением посредством обратной связи. Как было показано на примерах из реальной жизни, начиная с настольной игры Го и заканчивая манипуляциями физическими объектами, глубокое обучение с подкреплением позволяет машинам обрабатывать огромные объемы данных и выполнять осмысленные последовательности действий для решения сложных задач, что позволяет сопоставлять их с популярными представлениями об искусственном интеллекте.

[1] В 2015 году писатель и иллюстратор Тим Урбан (Tim Urban) представил серию статей из двух частей, которые в увлекательной форме описывают искусственный суперинтеллект и приводят ссылки на литературу. Статьи доступны по адресу bit.ly/urbanAI. (Перевод на русский язык можно найти по адресу https://habr.com/ru/post/293156/. — *Примеч. пер.*)

II

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ В КАРТИНКАХ

5

ТЕЛЕГА (КОД) ВПЕРЕДИ ЛОШАДИ (ТЕОРИИ)

В первой части книги был представлен общий обзор сферы глубокого обучения и продемонстрированы самые разные его применения. Попутно мы познакомились с основными понятиями глубокого обучения, от его иерархической природы обучения представлениям до связи с искусственным интеллектом. Также в процессе знакомства с основными идеями неоднократно отмечалось, что во второй части книги мы углубимся в базовую теорию и математический фундамент, стоящие за ними. Все *так* и будет, но мы хотим воспользоваться последней возможностью и сделать рассказ немного увлекательнее, поэтому поставим телегу (программный код) перед лошадью (в данном случае перед теорией).

В этой главе мы подробно исследуем код блокнота с моделью нейронной сети. Вам придется смириться с нашим решением отложить описание теории, на которой основывается этот код. Мы считаем, что такой извилистый путь поможет лучше понять теорию в последующих главах: каждое теоретическое положение, представленное в этой части книги, будет восприниматься уже не как абстрактная идея, а как вполне конкретное понятие, облеченное в осязаемые строки прикладного кода.

ПОДГОТОВКА

Работать с примерами из этой книги будет проще, если вы познакомитесь с основами командной строки Unix. Для этого мы советуем прочитать При-

ложение А из обманчиво простой книги Зеда Шоу (Zed Shaw) «Learn Python the Hard Way»[1,2].

Мы не случайно упомянули Python, потому что он является наиболее популярным языком программирования в сообществе специалистов по данным (по крайней мере, он был таковым на момент написания этих строк), а также потому, что мы выбрали его для примеров кода в этой книге. Python широко используется для разработки от простых автономных сценариев до больших моделей машинного обучения в промышленных системах. Если вы не знакомы с языком Python или чувствуете, что многое уже подзабыли, книга Шоу послужит вам неплохим общим справочником, а книга «Pandas for Everyone»[3] Дэниела Чена (Daniel Chen) идеально подойдет для изучения приемов применения языка в моделировании данных.

УСТАНОВКА

Независимо от того, планируете ли вы опробовать наши блокноты с кодом в Unix, Linux, macOS или Windows, мы подготовили пошаговые инструкции по установке, которые вы найдете в репозитории GitHub с примерами для этой книги:

github.com/the-deep-learners/deep-learning-illustrated

Предпочитающие просматривать готовые блокноты, а не запускать их на своем компьютере, найдут их в том же репозитории GitHub.

Мы решили демонстрировать примеры кода в форме удобных интерактивных блокнотов Jupyter[4]. В настоящее время Jupyter широко используется для разработки сценариев и обмена ими, особенно на этапах изысканий, когда исследователь экспериментирует с предварительной обработкой, визуализацией и моделированием своих данных. В наших инструкциях по установке рекомендуется запускать Jupyter в контейнере Docker[5]. Такой подход гарантирует наличие всех программных зависимостей, необходимых для запуска блокнотов, и в то же время предотвращает конфликт этих зависимостей с программным обеспечением, уже установленным в вашей системе.

---

[1] *Shaw Z.* (2013). Learn Python the Hard Way, 3rd Ed. New York: Addison-Wesley. Это приложение под названием «Command Line Crash Course» доступно онлайн по адресу learnpythonthehardway.org/book/appendixa.html.

[2] *Зед Шоу*. Легкий способ выучить Python. Бомбора, Эксмо, 2019, ISBN: 978-5-699-98251-6. — *Примеч. пер.*

[3] *Chen D.* (2017). Pandas for Everyone: Python Data Analysis. New York: Addison-Wesley.

[4] jupyter.org. Мы советуем познакомиться с горячими комбинациями клавиш для управления блокнотами Jupyter.

[5] docker.com

НЕГЛУБОКАЯ СЕТЬ В KERAS

Чтобы начать исследовать примеры программного кода в этой книге, мы:

1. Поближе познакомимся с широко известной коллекцией изображений рукописных цифр.
2. Загрузим эту коллекцию в блокнот Jupyter.
3. С помощью Python подготовим данные к моделированию.
4. Напишем несколько строк кода, использующих Keras, — высокоуровневую библиотеку глубокого обучения (сама она опирается на TensorFlow), чтобы создать искусственную нейронную сеть, которая предсказывает, какую цифру представляет данное изображение.

КОЛЛЕКЦИЯ ИЗОБРАЖЕНИЙ РУКОПИСНЫХ ЦИФР MNIST

В главе 1, представляя архитектуру компьютерного зрения LeNet-5 (рис. 1.11), мы упомянули, что одним из преимуществ, которые Ян Лекун (рис. 1.9) и его коллеги имели перед предыдущими практиками глубокого обучения, был превосходный набор данных. Этот набор изображений рукописных цифр с названием MNIST (см. примеры на рис. 5.1) упоминался также в контексте генеративно-состязательной сети Яна Гудфеллоу (рис. 3.2, а). Набор данных MNIST широко используется в руководствах по глубокому обучению, и тому есть веские причины. По современным меркам набор данных достаточно мал, и его можно быстро смоделировать даже на ноутбуке с не самым мощным процессором. Кроме небольшого размера набор MNIST также наглядно демонстрирует сложность классификации изображений: образцы рукописных цифр достаточно разнообразны и содержат довольно мелкие детали, распознавание которых вызывает сложности в алгоритмах машинного обучения, но все же не является непреодолимой задачей. Как вы увидите сами в этой части книги, хорошо спроектированная модель глубокого обучения может почти безошибочно распознавать рукописные цифры.

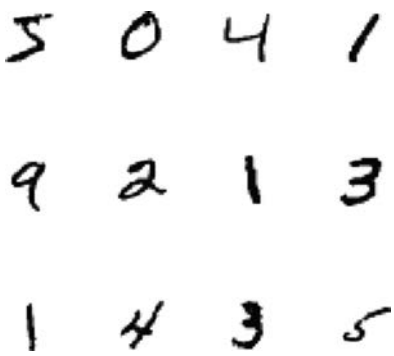


Рис. 5.1. Дюжина примеров изображений из коллекции MNIST. Каждое представляет единственную цифру, написанную рукой ученика средней школы или работника службы переписи населения США

Изображения для набора данных MNIST отбирались в 1990-х годах[1] Лекуном (см. рис. 1.9), Коринной Кортес (Corinna Cortes, рис. 5.2) и Крисом Берджесом (Chris Burges), исследователем ИИ из Microsoft. Этот набор включает 60 000 рукописных цифр для обучения алгоритмов и 10 000 для тестирования. Набор MNIST представляет собой подмножество более объемной коллекции образцов почерка учащихся средних школ и работников службы переписи населения США, собранной Национальным институтом стандартов и технологий США (National Institute of Standards and Technology, NIST).

Как показано на рис. 5.3, каждая цифра в наборе MNIST представлена изображением 28 × 28 пикселов[2]. Каждый пиксел — это 8-битное число, определяющее градацию серого цвета, то есть может изменяться в пределах от 0 (белый) до 255 (черный).

СХЕМА СЕТИ

В блокноте *shallow_net_in_keras.ipynb*[3] для Jupyter мы создаем искусственную нейронную сеть, которая определяет, какую цифру представляет данное изображение из набора MNIST. Как показано на рис. 5.4, эта искусственная нейронная сеть имеет только один скрытый слой искусственных нейронов и содержит всего три слоя. Согласно схеме на рис. 4.2, сети с таким небольшим количеством слоев обычно не принято считать архитектурами *глубокого* обучения, то есть это *неглубокая* сеть.

Рис. 5.2. Датский ученый-информатик Коринна Кортес — научный руководитель в Нью-Йоркском офисе Google. Помимо огромного вклада в теорию и практику машинного обучения, Кортес (с Крисом Берджесом и Яном Лекуном) курировала создание широко известного набора данных MNIST

Первый слой сети зарезервирован для ввода цифр MNIST. Поскольку они представлены изображениями размером 28 × 28 пикселов, каждая имеет 784 значения. После загрузки изображения преобразуются из двумерной формы 28 × 28 в одномерный массив с 784 элементами.

---

[1] yann.lecun.com/exdb/mnist/

[2] В языке Python отсчет индексов начинается с нуля, поэтому первая строка и столбец обозначены `0`, а 28-я строка и 28-й столбец обозначены как `27`.

[3] В папке *notebooks*, в репозитории GitHub с примерами для этой книги.

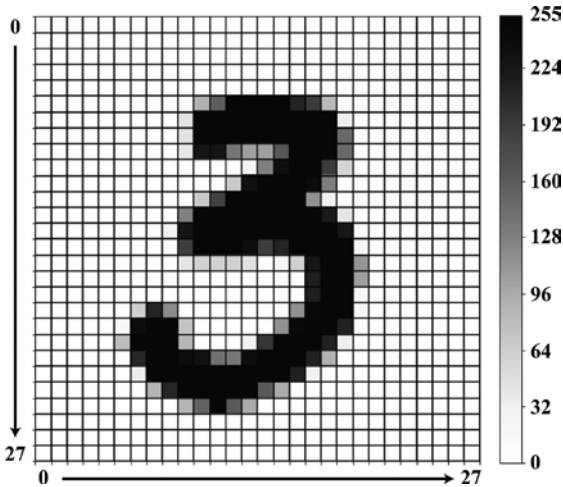


Рис. 5.3. Каждая рукописная цифра в наборе MNIST представлена черно-белым изображением размером 28 × 28 пикселов. Код, с помощью которого был создан этот рисунок, вы найдете в блокноте mnist_digit_pixel_by_pixel.ipynb, в репозитории с примерами для этой книги

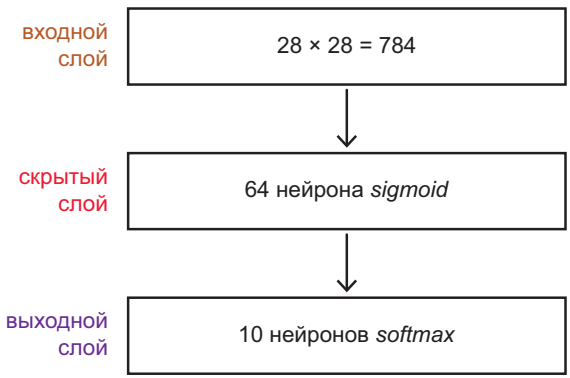


Рис. 5.4. Упрощенная схема поверхностной искусственной нейронной сети, которая рассматривается в этой главе. Подробнее о разновидностях нейронов sigmoid и softmax рассказывается в главах 6 и 7 соответственно

Входные данные с информацией о пикселах будут проходить через один скрытый слой[1], состоящий из 64 искусственных нейронов. Число (64) и тип (*sigmoid*) этих нейронов пока не важны для нас; подробнее эти атрибуты модели будут описаны в следующей главе. Самым главным на данный момент является тот факт (как было показано в главе 1, см. рис. 1.18 и 1.19), что нейроны в скрытом

[1] «Скрытые» слои называются так, потому что они не видны снаружи; входные данные влияют на них только косвенно, через входной или выходной слой нейронов.

слое отвечают за изучение представлений входных данных, чтобы сеть могла предсказать, какую цифру представляет данное изображение.

Можно утверждать, что преобразование двумерных изображений в одномерный массив приведет к потере значимой структуры рукописных цифр. И это верно! Однако, работая с одномерными данными, мы можем использовать относительно простые модели нейронных сетей, что как нельзя кстати на этом раннем этапе нашего путешествия. Позже, в главе 10, вы увидите более сложные модели, способные обрабатывать многомерные входные данные.

Информация, сгенерированная скрытым слоем, будет передана 10 нейронам в выходном слое. Мы подробно опишем работу нейронов *softmax* в главе 7, но суть в том, что 10 нейронов представляют 10 категорий цифр. Каждый из этих 10 нейронов сообщает вероятность, по одной для каждой из 10 возможных цифр, что данное изображение из коллекции MNIST представляет соответствующую цифру. Например, хорошо обученная сеть, получив изображение, показанное на рис. 5.3, может сообщить, что это изображение с вероятностью 0.92 представляет цифру *три*, с вероятностью 0.06 представляет цифру *два*, с вероятностью 0.02 представляет цифру *восемь* и с вероятностью 0 представляет остальные семь цифр.

ЗАГРУЗКА ДАННЫХ

Код в начале блокнота (листинг 5.1) импортирует программные зависимости, что является малоинтересным, но необходимым шагом.

Листинг 5.1. Программные зависимости, необходимые для построения неглубокой сети с помощью Keras

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot as plt
```

Здесь мы импортируем библиотеку Keras, с помощью которой будем создавать нашу нейронную сеть. Также мы импортируем набор MNIST — данные, которые обрабатывает этот пример. Назначение строк, оканчивающихся словами `Sequential`, `Dense` и `SGD`, мы объясним позже; вам пока не нужно беспокоиться о них. Наконец, строка `matplotlib` импортирует модуль, который поможет отобразить цифры MNIST на экране.

Импортировав зависимости, можно загрузить данные MNIST одной строкой кода, как показано в листинге 5.2.

Листинг 5.2. Загрузка данных MNIST

```
(X_train, y_train), (X_valid, y_valid) = mnist.load_data()
```

Теперь исследуем эти данные. Как упоминалось в главе 4, для представления данных, которые вводятся в модель, используется математическая нотация x, а для представления меток, используемых при обучении модели, используется нотация y. То есть `X_train` хранит изображения цифр из коллекции MNIST, на которых будет обучаться наша модель[1]. Обратившись к свойству `X_train.shape`, мы получим кортеж `(60000, 28, 28)`. Он показывает, что, как и ожидалось, в обучающем наборе данных содержится 60 000 изображений, каждое из которых представлено матрицей значений 28 × 28. Обратившись к свойству `y_train.shape`, можно обнаружить, что у нас имеется 60 000 меток, указывающих, какая цифра представлена каждым из 60 000 обучающих изображений. `y_train[0:12]` выведет массив с 12 целыми числами, представляющими первую дюжину меток, откуда можно понять, что первое изображение в обучающем наборе (`X_train [0]`) представляет цифру *пять*, второе — *ноль*, третье — *четыре* и т. д.

```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5], dtype=uint8)
```

Это та же дюжина цифр MNIST, которые показаны на рис. 5.1, созданная следующим фрагментом кода:

```
plt.figure(figsize=(5,5))
for k in range(12):
    plt.subplot(3, 4, k+1)
    plt.imshow(X_train[k], cmap='Greys')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Запросив аналогично форму проверочных данных (`X_valid.shape`, `y_valid.shape`), можно увидеть, что мы имеем ожидаемые 10 000 проверочных изображений с размером 28 × 28 пикселов, каждому их которых соответствует своя метка: `(10000, 28, 28)`, `(10000)`. Исследуя значения, составляющие отдельные изображения, такие как `X_valid[0]`, видно, что матрицы целых чисел, представляющие рукописные цифры, в основном состоят из нулей (белых пикселов). Наклонив голову вбок, можно даже заметить, что матрица в этом примере определяет цифру *семь*, где самые большие целые числа (например, 254, 255) соответствуют черному следу от ручки, который окружает контур из промежуточных значений, уменьшающихся до нуля при переходе к белой

[1] По общепринятым соглашениям имена переменных, хранящих двумерную матрицу или структуру данных с еще большей размерностью, начинаются с заглавной буквы, такой как `X`. А имена переменных, хранящих единственное значение (скаляр) или одномерный массив, начинаются со строчной буквы, такой как `x`.

бумаге. Чтобы убедиться, что это действительно число *семь*, мы вывели изображение командой `plt.imshow (X_valid [0], cmap = 'Grays')` (см. рис. 5.5) и его метку командой `y_valid [0]` (это оказалось число `7`).

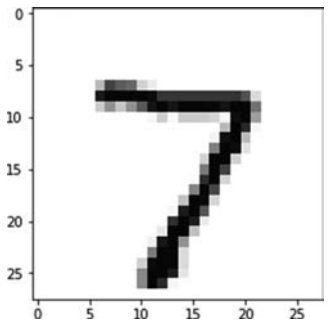


Рис. 5.5. Первая цифра в проверочном наборе из коллекции MNIST (`X_valid[0]`) — семь

ПЕРЕФОРМАТИРОВАНИЕ ДАННЫХ

Итак, данные загружены, и мы подошли к заголовку Preprocess data (Предварительная обработка данных) в блокноте. Но мы не будем обрабатывать изображения, применяя функции, скажем, для извлечения признаков, которые будут помогать обучаться нашей искусственной нейронной сети. Мы просто изменим форму данных, чтобы они соответствовали входному и выходному слою сети.

Проще говоря, мы преобразуем изображения размером 28 × 28 пикселов в одномерные массивы с 784 элементами в каждом. Это преобразование выполняет метод `reshape()`, как показано в листинге 5.3.

Листинг 5.3. Преобразование двумерных изображений в одномерные массивы

```
X_train = X_train.reshape(60000, 784).astype('float32')
X_valid = X_valid.reshape(10000, 784).astype('float32')
```

Одновременно используем `astype('float32')`, чтобы преобразовать целочисленные значения темных пикселов в значения с плавающей точкой одинарной точности[1]. Это преобразование является подготовкой к выполнению следую-

[1] Данные изначально хранятся как значения типа `uint8` — целые числа без знака в диапазоне от 0 до 255. Этот формат позволяет экономно расходовать память, но не обеспечивает большой точности, потому что поддерживает только 256 возможных значений. Если не указать явно, интерпретатор Python по умолчанию будет использовать 64-разрядные числа с плавающей точкой, что в данном случае излишне. Поэтому, выбрав 32-разрядное представление значений с плавающей точкой, мы намеренно уменьшили точность представления значений, не оказав отрицательного влияния на работу примера.

щего шага, показанного в листинге 5.4, где мы делим все значения на 255, чтобы отобразить их в диапазоне значений от 0 до 1[1].

Листинг 5.4. Преобразование целочисленных значений пикселов в вещественные

```
X_train /= 255
X_valid /= 255
```

Вернемся к нашему примеру с рукописной цифрой *семь* на рис. 5.5: если выполнить `X_valid[0]`, можно убедиться, что теперь это одномерный массив, состоящий из значений с плавающей точкой от 0 до 1.

Это результат преобразования входных данных X модели. Как показано в листинге 5.5, метки y тоже нужно преобразовать из целых чисел в прямой код (что это такое, будет показано ниже на практическом примере).

Листинг 5.5. Преобразование целочисленных меток в прямой код

```
n_classes = 10
y_train = keras.utils.to_categorical(y_train, n_classes)
y_valid = keras.utils.to_categorical(y_valid, n_classes)
```

Есть всего 10 возможных рукописных цифр, поэтому переменная `n_classes` должна быть равной `10`. В двух других строках вызывается вспомогательная функция `to_categorical` из библиотеки Keras, преобразующая метки в обучающем и проверочном наборах данных из целочисленного представления в прямой код. Выполним `y_valid`, чтобы увидеть, как теперь выглядит метка *семь*:

```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

Теперь вместо целого числа *семь* мы имеем массив с 10 элементами, состоящий полностью из `0`, кроме `1` в восьмой позиции. В таком прямом коде метка *ноль* будет представлена массивом с единственной `1` в первой позиции, метка *один* — с единственной `1` во второй позиции и т. д. В таком прямом коде метки представляют 10 вероятностей, что выводятся последним слоем нашей искусственной нейронной сети. Они описывают идеальный результат, которого мы стремимся достичь: если входное изображение представляет собой рукописную цифру *семь*, тогда хорошо обученная сеть должна сообщить вероятность 1.00, что это *семь*, и вероятность 0.00 для всех остальных девяти цифр.

---

[1] Модели машинного обучения, как правило, учатся более эффективно, если передавать им данные в стандартизированном виде. Бинарные входные данные обычно передаются в виде значений `0` и `1`, тогда как распределения часто нормализуются и приводятся к распределению со средним значением 0 и стандартным отклонением 1. В данном случае мы привели интенсивность окраски пикселов к диапазону от 0 до 1.

ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ НЕЙРОННОЙ СЕТИ

С нашей точки зрения, это самая интересная часть сценария с реализацией глубокого обучения: создание самой искусственной нейронной сети. Многообразие архитектур бесконечно, и по мере чтения книги вы будете развивать интуицию, которая поможет выбирать архитектуры для своих экспериментов в той или иной сфере. Как показано на рис. 5.4, мы выбрали наиболее простую архитектуру, реализация которой показана в листинге 5.6.

Листинг 5.6. Реализация архитектуры неглубокой нейронной сети с использованием Keras

```
model = Sequential()
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

Первая строка кода создает объект модели нейронной сети простейшего типа — `Sequential`[1], и, испытывая небывалый прилив вдохновения, мы дали объекту модели имя `model`. Вторая строка использует метод `add()` объекта `model`, чтобы определить атрибуты скрытого слоя сети (`64` искусственных нейрона типа `sigmoid` в полносвязанном слое, который в свою очередь создается методом `Dense()`)[2], а также форму входного слоя (одномерный массив с длиной `784`). В третьей и последней строке снова используется метод `add()`, чтобы добавить выходной слой и определить его параметры: `10` искусственных нейронов типа `softmax`, соответствующих 10 вероятностям (по одной на каждую из 10 возможных цифр), через которые сеть будет выводить результат после передачи ей изображения рукописной цифры.

ОБУЧЕНИЕ МОДЕЛИ ГЛУБОКОГО ОБУЧЕНИЯ

Позже мы еще вернемся к шагам `model.summary()` и `model.compile()` в блокноте *shallow_net_in_keras.ipynb*, а также к трем строкам с арифметическими выражениями. А сейчас перейдем к этапу обучения модели (листинг 5.7).

Листинг 5.7. Обучение неглубокой нейронной сети с использованием Keras

```
model.fit(X_train, y_train,
          batch_size=128, epochs=200,
          verbose=1,
          validation_data=(X_valid, y_valid))
```

[1] Такое название этот тип моделей получил потому, что каждый слой сети *последовательно* (sequential) передает информацию только следующему слою.

[2] Повторим еще раз, что все эти необычные термины мы поясним в следующих главах.

Наиболее важные аспекты:

1. Метод `fit()` объекта `model` запускает обучение сети на обучающих изображениях `X_train` — входных данных — и связанных с ними метках `y_train` — желаемых выходных данных.
2. Также `fit()` дает возможность оценивать точность прогнозов сети в ходе ее обучения, для чего достаточно передать проверочные данные `X_valid` и `y_valid` в аргументе `validation_data`.
3. В машинном и особенно в глубоком обучении обычной практикой является многократное обучение модели на одних и тех же данных. Один проход через все обучающие данные (60 000 изображений в данном случае) называется *эпохой* обучения. Присвоив параметру `epochs` значение `200`, мы требуем перебрать все 60 000 обучающих изображений 200 раз.
4. Передав в параметре `verbose` число `1`, мы потребовали от метода `model.fit()` выводить подробный отчет о ходе обучения. В данный момент нас интересует статистика `val_acc`, которая выводится после каждой эпохи обучения. `val_acc` (*validation accuracy — точность на проверочных данных*) — это доля из 10 000 рукописных изображений в `X_valid`, для которых сеть правильно указала наибольшую вероятность для цифры, согласно меткам в `y_valid`.

Как можно видеть в блокноте, после первой эпохи обучения оценка `val_acc` получила значение `0.1010`, то есть 10.1% изображений из проверочного набора данных были правильно классифицированы нашей неглубокой сетью. Учитывая, что всего имеется 10 классов рукописных цифр, можно предположить, что случайное угадывание тоже даст точность около 10%, поэтому полученный результат не впечатляет. Однако по мере дальнейшего обучения сети результаты улучшаются. После 10 эпох обучения она правильно классифицирует 36.5% проверочных изображений — намного лучше, чем может дать случайное угадывание! И это только начало: после 200 эпох возможности сети, по всей видимости, подошли к пределу, потому что показатель точности все медленнее и медленнее приближается к отметке в 86%. Учитывая, что мы создали очень простую, поверхностную нейронную сеть, это не так уж плохо!

ИТОГИ

Поставив телегу впереди лошади, в этой главе мы реализовали неглубокую и очень простую искусственную нейронную сеть. Она может со вполне приличной точностью классифицировать изображения из коллекции MNIST. В остальных главах второй части книги, по мере углубления в теорию, мы познакомимся с накопленным опытом создания искусственных нейронных сетей и перейдем к аутентичным архитектурам глубокого обучения. С их помощью у нас наверняка получится классифицировать входные данные гораздо точнее, или не получится? Давайте посмотрим.

6

ИСКУССТВЕННЫЕ НЕЙРОНЫ, ОПРЕДЕЛЯЮЩИЕ ХОТ-ДОГИ

После знакомства с областями применения глубокого обучения в первой части книги и с функционирующей нейронной сетью в главе 5 можно приступать к изучению теории, лежащей в основе всего этого. Первыми мы исследуем искусственные нейроны — минимальные элементы, которые при объединении образуют искусственную нейронную сеть.

ВВЕДЕНИЕ В БИОЛОГИЧЕСКУЮ НЕЙРОАНАТОМИЮ

Как было показано во вступительных абзацах этой книги, искусственные нейроны были созданы по аналогии с биологическими. Учитывая это, рассмотрим рис. 6.1, который используется для демонстрации на первой лекции в любом курсе нейроанатомии. Здесь изображен биологический нейрон, получающий входные сигналы от множества (как правило, нескольких тысяч) *дендритов*, причем каждый дендрит получает информационный сигнал от другого нейрона в нервной системе — так образуется биологическая нейронная сеть. Когда сигнал, передаваемый по дендриту, достигает тела клетки, он вызывает небольшое изменение ее электрического потенциала[1]. Одни дендриты вызывают небольшое положительное изменение потенциала, другие — отрицательное. Если совокупный эффект этих изменений вызывает увеличение потенциала покоя –70 милливольт до критического порога –55 милливольт, называемого *потенциалом действия*, нейрон инициирует волну возбуждения вниз от тела клетки по аксону, передавая таким образом сигнал другим нейронам в сети.

[1] Точнее, разность электрических потенциалов на внутренней и наружной сторонах мембраны клетки.

В итоге в биологических нейронах наблюдаются следующие три стадии распространения сигнала:

1. **Прием информации** от множества других нейронов.
2. **Агрегирование полученной информации** изменением потенциала тела клетки.
3. **Передача сигнала, если электрический потенциал клетки превысил пороговый уровень.** Этот сигнал может быть получен многими другими нейронами в сети.

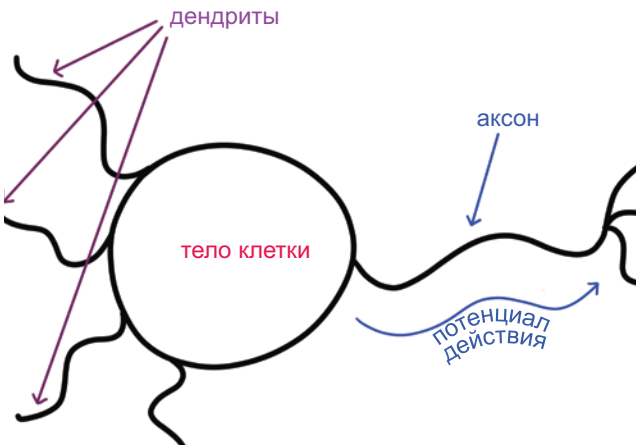


Рис. 6.1. Анатомия биологического нейрона

Мы выделили фиолетовым, красным и синим цветом текст, соответствующий элементам нейрона на рис. 6.1 (дендриты, тело клетки и аксон соответственно). Эти же цвета мы будем использовать снова и снова на протяжении всей книги, в том числе и в обсуждении ключевых уравнений и переменных, содержащихся в них.

ПЕРЦЕПТРОН

В конце 1950-х годов американский нейробиолог Фрэнк Розенблатт (Frank Rosenblatt, рис. 6.2) опубликовал статью о своем *перцептроне*, алгоритме, основанном на его понимании биологических нейронов, что можно считать самой первой попыткой реализовать искусственный нейрон[1]. Подобно своему биологическому аналогу, перцептрон (рис. 6.3) может:

[1] *Rosenblatt F.* (1958). «The perceptron: A probabilistic model for information storage and the organization in the brain». Psychological Review, 65, 386–408.

1. **Принимать информацию** от множества других нейронов.
2. **Агрегировать эту информацию** с использованием простой арифметической операции, которая называется *взвешенной суммой*.
3. **Генерировать выходной сигнал**, если взвешенная сумма превысит пороговое значение, который затем может быть отправлен многим другим нейронам в сети.

ДЕТЕКТОР ХОТ-ДОГОВ

Разберем на простеньком примере, как работает алгоритм перцептрона. Возьмем перцептрон, который специализируется на различении объектов: он определяет, является ли объект хот-догом или, эм-м... не хот-догом.

Рис. 6.2. Американский исследователь в области нейробиологии и поведения Франк Розенблатт. Большую часть жизни работал в Корнеллской авиационной лаборатории и там же сконструировал свой первый физический перцептрон Mark I Perceptron. Эту машину, ранний реликт искусственного интеллекта, сегодня можно увидеть в Смитсоновском институте в Вашингтоне, округ Колумбия

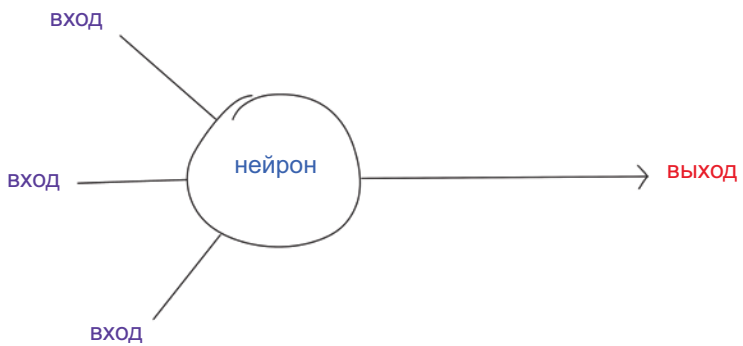


Рис. 6.3. Принципиальная схема перцептрона, раннего искусственного нейрона. Обратите внимание на структурное сходство с биологическим нейроном на рис. 6.1

Важнейшей особенностью перцептронов является их способность принимать и передавать только двоичную информацию. То есть наш перцептрон — детектор хот-догов — должен получать три входных сигнала (указывающих

наличие кетчупа, горчицы и булочки в рассматриваемом объекте) в виде 0 или 1. На рис. 6.4:

- первый вход (фиолетовый, со значением 1) сообщает перцептрону о наличии кетчупа в составе объекта;
- второй вход (тоже фиолетовый, со значением 1) сообщает о наличии горчицы;
- третий вход (фиолетовый, со значением 0) сообщает об *отсутствии* булочки.

Чтобы предсказать, является ли объект хот-догом или нет, перцептрон независимо *взвешивает* каждый из этих трех входов[1]. Веса, которые мы произвольно выбрали в этом примере, показывают, что наличие булочки (вес 6) является наиболее существенным признаком хот-дога. Вторым по значимости (с весом 3) является наличие кетчупа, а наименее значимым (с весом 2) — наличие горчицы.

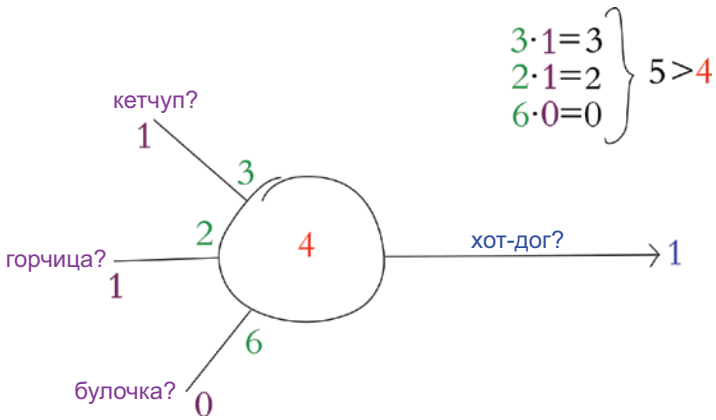


Рис. 6.4. Первый пример перцептрона, отличающего хот-доги: в данном примере он приходит к выводу, что объект является хот-догом

Давайте определим взвешенную сумму входных данных: каждый вход по отдельности (то есть поэлементно) умножим на его вес и затем сложим получившиеся результаты. Сначала посчитаем взвешенные входные значения:

1. Для входа кетчуп: $3 \times 1 = 3$.
2. Для горчицы: $2 \times 1 = 2$.
3. Для булочки: $6 \times 0 = 0$.

[1] Эта парадигма должна быть хорошо знакома всем, имеющим опыт регрессионного моделирования.

Получив три произведения, вычисляем взвешенную сумму входных данных и получаем 5: 3 + 2 + 0. В общем случае расчет взвешенной суммы входных данных вычисляется так:

$$\sum_{i=1}^{n} w_i x_i, \tag{6.1}$$

где:

- w_i — вес i-го входа (в данном примере: $w_1 = 3$, $w_2 = 2$ и $w_3 = 6$);
- x_i — значение i-го входа (в данном примере: $x_1 = 1$, $x_2 = 1$ и $x_3 = 0$);
- $w_i x_i$ — произведение w_i и x_i, то есть взвешенное значение i-го входа;
- $\sum_{i=1}^{n}$ — обозначает сумму всех взвешенных входных значений $w_i x_i$, где n — общее количество входов (в данном примере у нас три входа, но вообще искусственные нейроны могут иметь любое количество входов).

Заключительный шаг алгоритма перцептрона — сравнение взвешенной суммы входов с *порогом* нейрона. По аналогии с весами мы произвольно выбрали пороговое значение *4* для нашего примера перцептрона (выделено красным в центре нейрона на рис. 6.4). То есть в общем виде алгоритм перцептрона выглядит так:

$$\sum_{i=1}^{n} w_i x_i \begin{cases} > \text{ порогового значения, вывести 1} \\ \leq \text{ порогового значения, вывести 0,} \end{cases} \tag{6.2}$$

где:

- если взвешенная сумма входных значений перцептрона превышает *пороговое значение*, выводится 1, то есть перцептрон предсказывает, что объект является хот-догом;
- если взвешенная сумма входных значений перцептрона меньше или равна *пороговому значению*, выводится 0, то есть перцептрон предсказывает, что объект *не* является хот-догом.

Зная это, мы можем сделать следующий окончательный вывод в нашем примере на рис. 6.4: взвешенная сумма 5 больше *порогового значения* 4, поэтому перцептрон определяет объект как хот-дог и выводит 1.

Теперь передадим перцептрону объект на рис. 6.5, содержащий только горчицу — без кетчупа и без булочки, — и оценим его, следуя логике первого примера. В этом случае взвешенная сумма входных данных равна `2`. Поскольку `2` меньше *порога* перцептрона, нейрон выведет `0`, предсказывая, что этот объект не является хот-догом.

В третьем и заключительном примере, показанном на рис. 6.6, искусственный нейрон оценивает объект без горчицы и кетчупа, но *с* булочкой. Наличие одной только булочки дает взвешенную сумму `6`. Поскольку `6` больше *порога* перцептрона, алгоритм предскажет, что объект является хот-догом, и выведет `1`.

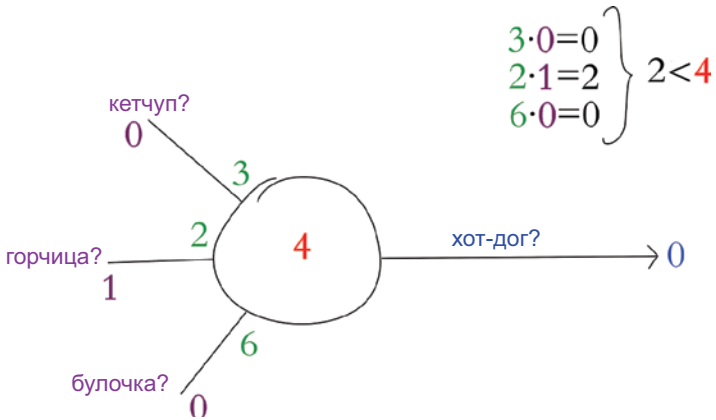


Рис. 6.5. Второй пример перцептрона, отличающего хот-доги: в данном примере он приходит к выводу, что объект не является хот-догом

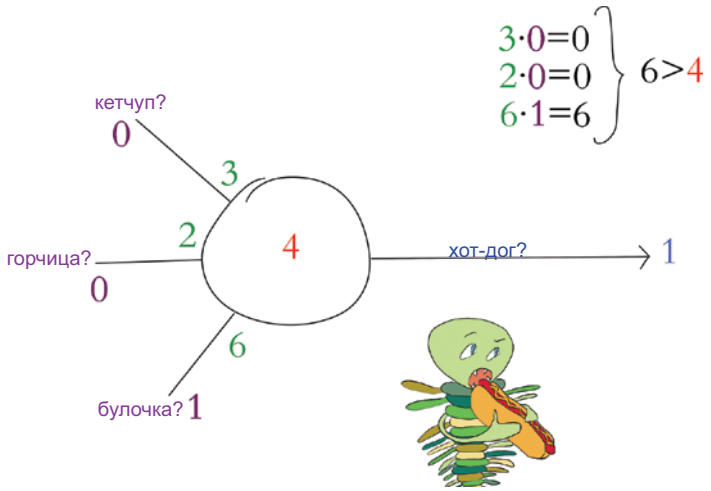


Рис. 6.6. Третий пример перцептрона, отличающего хот-доги: в данном примере он приходит к выводу, что объект является хот-догом

САМОЕ ВАЖНОЕ УРАВНЕНИЕ В ЭТОЙ КНИГЕ

Чтобы сформулировать упрощенное и универсальное уравнение перцептрона, введем понятие *смещения* (*bias*), которое обозначим как b и будем считать эквивалентным отрицательному значению *порогового значения* искусственного нейрона:

$$b \equiv -\textit{пороговое значение}. \tag{6.3}$$

Смещение и веса нейрона вместе являются его *параметрами* — изменяемыми переменными, которые определяют, что нейрон выведет при получении тех или иных входных данных.

Теперь, определив понятие смещения нейрона, мы приходим к наиболее широко используемому уравнению перцептрона:

$$\text{выход} \begin{cases} 1, & \text{если } w \cdot x + b > 0 \\ 0 & \text{в остальных случаях.} \end{cases} \tag{6.4}$$

Обратите внимание на пять следующих изменений в исходном уравнении перцептрона (6.2):

1. На место *порогового значения* нейрона мы подставили смещение b.
2. Перенесли b в правую сторону уравнения, поместив рядом с остальными переменными.
3. Использовали массив w для представления всех весов, от w_1 до w_n.
4. Аналогично использовали массив x для представления всех значений x_i, от x_1 до x_n.
5. Использовали нотацию скалярного произведения $w \cdot x$ для сокращенного представления вычисления взвешенной суммы всех входов нейрона (более длинная форма записи этого произведения показана в уравнении 6.1: $\sum_{i=1}^{n} w_i x_i$).

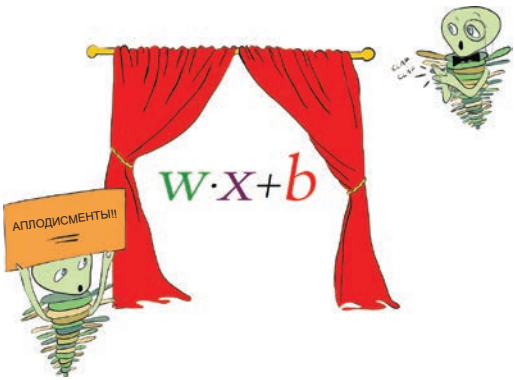


Рис. 6.7. Обобщенное уравнение для искусственных нейронов, к которому мы будем возвращаться снова и снова, — самое важное уравнение в этой книге

В основе уравнения 6.4 перцептрона лежит выражение $w \cdot x + b$, которое мы выделили на рис. 6.7, чтобы акцентировать внимание, и поместили отдельно. *Важный урок, который вы должны вынести из этой главы: запомните, что эта формула с тремя переменными представляет искусственные нейроны в целом.* Мы часто будем ссылаться на это уравнение.

Чтобы максимально упростить арифметику в наших примерах перцептрона, отличающего хот-доги, все параметры — веса перцептрона и его смещение — имеют только положительные целочисленные значения. Однако на практике эти параметры могут иметь отрицательные значения и редко бывают целыми числами. Обычно они настраиваются как значения с плавающей точкой.

Наконец, всем параметрам в этих примерах были присвоены произвольно выбранные значения, тогда как обычно они корректируются в процессе обучения искусственных нейронов. В главе 8 мы рассмотрим, как эта корректировка параметров осуществляется на практике.

СОВРЕМЕННЫЕ НЕЙРОНЫ И ФУНКЦИИ АКТИВАЦИИ

Современные искусственные нейроны, как те, что находятся в скрытом слое неглубокой сети, построенной нами в главе 5 (посмотрите рис. 5.4 или наш блокнот *shallow_net_in_keras.ipynb*), не являются перцептронами. Перцептрон послужил нам относительно простым введением в искусственные нейроны, но в настоящее время он редко используется на практике. Наиболее очевидным ограничением перцептрона является бинарная природа его входов и выхода. Во многих случаях бывает желательно делать прогнозы на основе входных данных, полученных из непрерывных диапазонов значений, поэтому одно это ограничение делает перцептроны непригодными.

Менее очевидное (но еще более критичное) следствие бинарной природы перцептрона состоит в том, что этот аспект существенно осложняет обучение. Взгляните на рис. 6.8, где мы используем новый член z, представляющий значение уравнения $w \cdot x + b$ из рис. 6.7.

Когда z — любое значение, меньше нуля или равное ему, перцептрон выводит наименьшее возможное значение `0`. Если z получает положительное значение,

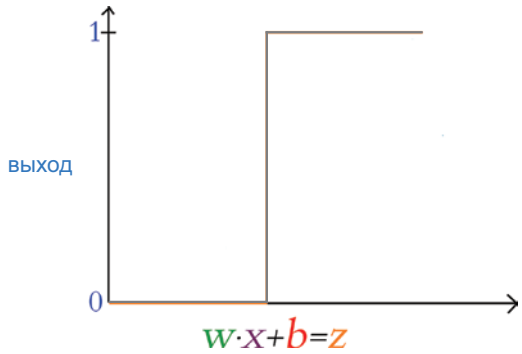


Рис. 6.8. Переход выхода перцептрона от нулевого к единичному значению происходит мгновенно, что затрудняет тонкую настройку w и b в соответствии с желаемым выходным значением

даже чуть больше нуля, перцептрон выводит наибольшее возможное значение 1. Этот мгновенный переход из одного крайнего значения в другое усложняет обучение: обучая сеть, мы вносим небольшие корректировки в w и b, которые, казалось бы, должны улучшить точность прогнозирования сети[1]. Для перцептрона большинство незначительных корректировок w и b не будут влиять на его выходное значение; z обычно колеблется вокруг отрицательных значений, близких к 0, или вокруг положительных значений, значительно превышающих 0. Такое поведение не только бесполезно, но и ухудшает ситуацию: время от времени происходит небольшая корректировка w или b, из-за которой z переходит из отрицательной области значений в положительную (или наоборот), что вызывает резкое изменение выходного значения с 0 до 1 (или наоборот). По сути, у перцептрона нет промежуточных состояний — он либо молчит, либо кричит во весь голос.

НЕЙРОНЫ SIGMOID

На рис. 6.9 представлена альтернатива неустойчивому поведению перцептрона: плавная кривая от 0 до 1. Эта конкретная форма кривой называется *сигмоидной* (sigmoid) функцией и определяется как

$$\sigma(z) = \frac{1}{1+e^{-z}},$$

где:

- z — это эквивалент выражения $w \cdot x + b$;
- e — это математическая константа 2.718..., она известна своей ролью в натуральной экспоненциальной функции;
- σ — греческая буква «сигма», корень слова «сигмоида».

Сигмоидная функция — наш первый пример *функции активации* искусственного нейрона. Возможно, вы помните о ней, потому что именно она была типом нейронов для скрытого слоя в нашей неглубокой нейронной сети в главе 5. Как вы увидите далее в этом разделе, сигмоидная функция является канонической функцией активации, причем настолько канонической, что для ее обозначения была выбрана греческая буква σ (сигма), традиционно используемая для обозначения любой функции активации.

Результат функции активации любого данного нейрона называется просто его *активацией*, и в этой книге мы будем обозначать этот член буквой a, например, рядом с вертикальной осью на рис. 6.9.

---

[1] Под *улучшением* здесь подразумевается получение выходного значения, которое точнее соответствует истинному выходному значению y для некоторого входного значения x. Мы обсудим этот вопрос в главе 8.

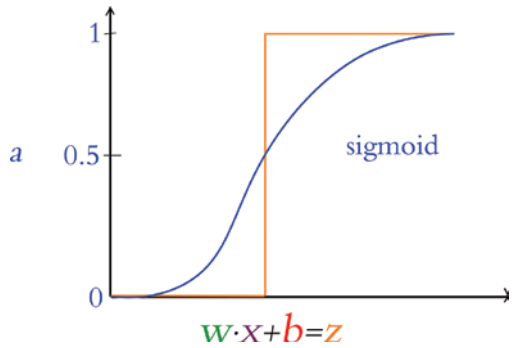


Рис. 6.9. Сигмоидная функция активации

На наш взгляд, не нужно стараться запомнить сигмоидную функцию (и вообще любую из функций активации). Вместо этого лучше поэкспериментировать с ее поведением в интерактивном режиме. Поэтому откройте блокнот Jupyter *sigmoid_function.ipynb* из репозитория GitHub с примерами для книги и следуйте за нашими объяснениями кода.

Единственная зависимость в блокноте — это константа *e*, которую мы загружаем с помощью инструкции `from math import e`. Далее начинается самое интересное — мы определяем саму сигмоидную функцию:

```
def sigmoid(z):
    return 1/(1+e**-z)
```

Как показано на рис. 6.9 и демонстрирует вызов `sigmoid(.00001)`, для входных значений, близких к `0`, сигмоидная функция возвращает значения около `0.5`. Чем больше будет положительное входное значение, тем ближе к `1` будет результат. В примере с входным значением `10000` функция вернула `1.0`. Постепенно уменьшая входные значения в отрицательном направлении, мы получаем на выходе числа, плавно приближающиеся к `0`: например, `sigmoid(-1)` возвращает `0.2689`, а `sigmoid(-10)` возвращает `4.5398e-05`[1].

Любой искусственный нейрон с сигмоидной функцией активации называется *сигмоидным нейроном* (нейрон типа sigmoid), и его преимущество перед перцептроном теперь должно быть очевидно: небольшие изменения параметров w и b такого сигмоидного нейрона вызывают небольшие изменения z и, соответственно, небольшие изменения в активации нейрона, a. Большие отрицательные или большие положительные значения z являются исключением: при экстремально больших по абсолютной величине значениях z сигмоидные

[1] Не путайте букву `e` в числе `4.5398e-05` с основанием натурального логарифма *e*. В числах, которые выводит программа, буква `e` обозначает *показатель степени (экспоненту)*, то есть данное выходное значение эквивалентно числу 4.5398×10^{-5}.

нейроны, как и перцептроны, будут выводить `0` (для отрицательных z) или `1` (для положительных z). Как и в случае с перцептроном, это означает, что небольшие изменения весов и смещений во время обучения мало влияют на результат, и, как следствие, обучение останавливается. Эта ситуация называется *насыщением* нейрона и может возникать при использовании большинства функций активации. К счастью, есть приемы, позволяющие избежать насыщения, как будет показано в главе 9.

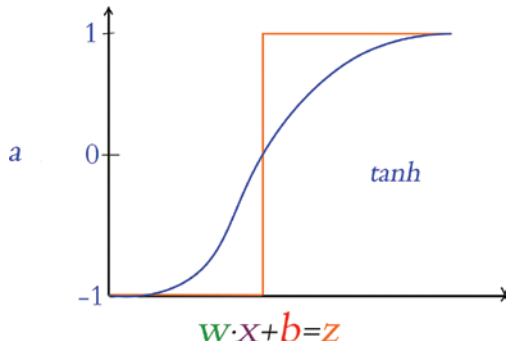


Рис. 6.10. Функция активации tanh

НЕЙРОН ТИПА TANH

Наряду с сигмоидным нейроном большой популярностью в среде практиков глубокого обучения пользуется его близкий родственник — нейрон типа *tanh* (произносится «танх»). Функция активации *tanh* показана на рис. 6.10 и определяется как

$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}.$$

Своей формой кривая *tanh* похожа на сигмоиду, но, в отличие от сигмоидной функции, которая имеет область значений [0:1], функция *tanh* имеет область значений [−1:1]. Эта разница носит больше косметический характер. Отрицательным входным значениям z соответствуют отрицательные активации a, $z = 0$ соответствует $a = 0$, и положительным z соответствуют положительные активации a, то есть серединой распределения выходных значений нейрона типа *tanh* является число 0. Как будет показано далее в главах с 7 по 9, такие выходы a с центром распределения в точке 0 обычно служат входами x для других искусственных нейронов в сети, а входы с центром распределения в точке 0 делают (подумать только!) насыщение нейронов менее вероятным, что позволяет сети учиться более эффективно.

RELU: RECTIFIED LINEAR UNIT

Последний тип нейронов, который мы рассмотрим, — это *блок линейной ректификации* (Rectified Linear Unit, ReLU), его поведение показано на рис. 6.11. Функция активации ReLU, форма которой резко отличается от формы функций sigmoid и tanh, была создана на основе свойств биологических нейронов[1] и популяризирована для применения в искусственных нейронных сетях Винодом Наиром (Vinod Nair) и Джеффом Хинтоном (см. рис. 1.16)[2]. Форма функции ReLU определяется уравнением $a = max\ (0, z)$.

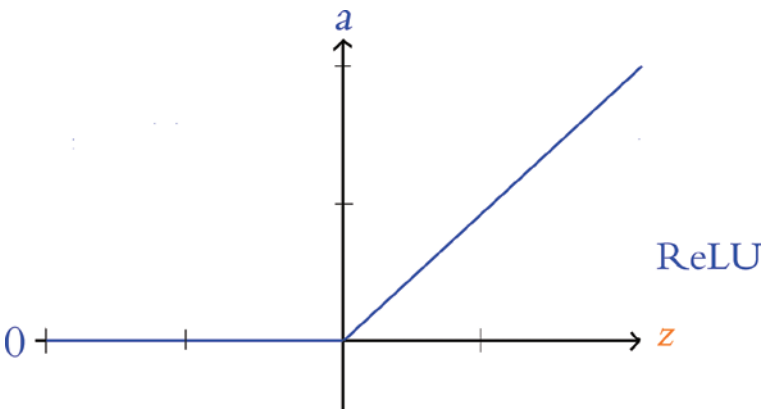


Рис. 6.11. Функция активации ReLU

Эта функция выражается просто:

- если z имеет положительное значение, функция активации ReLU возвращает z, то есть $a = z$;
- если $z = 0$ или имеет отрицательное значение, функция возвращает свое минимальное значение 0, то есть $a = 0$.

ReLU — одна из простейших функций, используемых для представления *нелинейности*. То есть ее результат a, так же как результат функций sigmoid и tanh, изменяется нелинейно по всем значениям z. По сути, ReLU состоит из *двух* линейных функций (одна, возвращающая 0, используется при отрицательных значениях z, а другая, возвращающая z, — при положительных, как показано на

[1] Потенциал действия биологических нейронов имеет только «положительный» режим запуска; у них нет «отрицательного» режима. См. Hahnloser, R., & Seung, H. (2000). «Permitted and forbidden sets in symmetric threshold-linear networks». Advances in Neural Information Processing Systems, 13.

[2] *Nair V. & Hinton G.* (2010). «Rectified linear units improve restricted Boltzmann machines». Proceedings of the International Conference on Machine Learning.

рис. 6.11), объединение которых формирует нелинейную функцию. Нелинейная природа является важным свойством всех функций активации, используемых в архитектурах глубокого обучения. Как показано в серии увлекательных интерактивных апплетов в главе 4 электронной книги Майкла Нильсена (Michael Nielsen) «Neural Networks and Deep Learning», нелинейность позволяет моделям глубокого обучения аппроксимировать любую непрерывную функцию[1]. Эта универсальная способность аппроксимировать результат y с учетом некоторого входного значения x является одним из признаков глубокого обучения — характеристики, которая делает этот подход очень эффективным в широком диапазоне приложений.

Относительно простая форма нелинейности функции ReLU является ее преимуществом. Как будет показано в главе 8, определение подходящих значений w и b в сетях глубокого обучения включает определение частных производных, а эти операции на линейных частях функции ReLU более эффективны в вычислительном отношении, чем на кривых, таких как функции sigmoid и tanh[2]. Как свидетельство их полезности, включение нейронов ReLU в AlexNet (рис. 1.17) стало одним из факторов, обеспечивших победу над существовавшими эталонами компьютерного зрения в 2012 году и развитие в эпоху глубокого обучения. На сегодняшний день нейроны ReLU наиболее широко используются в скрытых слоях глубоких искусственных нейронных сетей, и вы увидите их в большинстве блокнотов Jupyter, написанных для этой книги.

ВЫБОР ТИПА НЕЙРОНОВ

Определяя организацию скрытого слоя искусственной нейронной сети, можно выбрать любую понравившуюся функцию активации. Но чтобы модель глубокого обучения имела возможность аппроксимировать любую непрерывную функцию, это должна быть только нелинейная функция, однако даже с этим ограничением остается достаточно обширный выбор. Чтобы в будущем вы могли осознанно принять решение, оценим типы нейронов, которые мы обсудили в этой главе, расположив их в порядке от менее предпочтительных к более предпочтительным:

1. *Перцептрон* с его бинарными входами и резко изменяющимся бинарным выходом едва ли можно считать практичным выбором для моделей глубокого обучения.

---

[1] neuralnetworksanddeeplearning.com/chap4.html

[2] Кроме того, существует множество исследований, согласно которым активации ReLU поощряют *разреженность параметров*, то есть они являются наименее сложными функциями уровня нейронной сети, улучшающими обобщение проверочных данных. Более подробно об обобщающих способностях моделей будет рассказываться в главе 9.

2. *Сигмоидный* нейрон представляет более приемлемый вариант, но сети с нейронами этого типа обычно обучаются медленнее, чем сети со, скажем, tanh или ReLU. Поэтому мы советуем применять сигмоидные нейроны только в ситуациях, когда можно использовать тот факт, что его выходное значение изменяется в диапазоне [0, 1][1].
3. Нейрон *tanh* — хороший выбор. Как уже отмечалось выше, выход с центром распределения значений в точке 0 помогает глубоким сетям обучаться быстрее.
4. Мы сами предпочитаем нейроны *ReLU*, потому что они обеспечивают более высокую эффективность вычислений в алгоритмах обучения. Как показывает наш опыт, благодаря им искусственные нейронные сети получаются более точными и обучаются быстрее.

Кроме типов нейронов, описанных в этой главе, существует еще целый ряд функций активации, и их список постоянно растет. На момент написания этой книги библиотека Keras предлагала несколько «продвинутых» функций активации[2]: *ReLU с «утечкой»* (leaky ReLU), *параметрический ReLU* и *экспоненциально линейный блок* (Exponential Linear Unit, ELU), которые являются производными от ReLU. Мы советуем познакомиться с описанием этих активаций в документации к библиотеке Keras. Также вы можете поменять тип нейронов, используемый в любом из блокнотов Jupyter для этой книги, и сравнить результаты. Мы будем приятно удивлены, если вы обнаружите, что они обеспечивают более высокую эффективность или точность в ваших нейронных сетях, которые намного превзойдут наши.

ИТОГИ

В этой главе мы подробно описали математику нейронов, составляющих искусственные нейронные сети, включая модели глубокого обучения. Мы также перечислили плюсы и минусы наиболее известных типов нейронов и дали рекомендации, которыми вы сможете руководствоваться при выборе нейронов для своих моделей глубокого обучения. В главе 7 мы расскажем, как искусственные нейроны объединяются в сеть для изучения особенностей исходных данных и аппроксимации сложных функций.

---

[1] В главах 7 и 11 будет показана пара таких ситуаций, в частности с сигмоидным нейроном в качестве единственного нейрона в выходном слое сети бинарного классификатора.

[2] См. документацию к библиотеке Keras, доступную по адресу keras.io/layers/advanced-activations.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Двигаясь от главы к главе, мы будем постепенно добавлять новые термины в этот список ключевых понятий. Если вы запомните эти основополагающие концепции, вам проще будет понять последующие главы и к концу книги прочно овладеть теорией и практикой глубокого обучения. Вот новые важные понятия, представленные в этой главе:

- параметры:
  - вес w;
  - смещение b;
- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU.

7

ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

В главе 6 мы познакомились с некоторыми особенностями искусственных нейронов. Эта глава является естественным продолжением: здесь мы посмотрим, как отдельные нейроны связываются друг с другом, образуя искусственные нейронные сети, в том числе сети глубокого обучения.

ВХОДНОЙ СЛОЙ

В блокноте Jupyter *shallow_net_in_keras.ipynb* мы сконструировали неглубокую искусственную нейронную сеть (схема которой показана на рис. 5.4), использовав следующие слои:

1. *Входной* слой, состоящий из 784 нейронов, по одному для каждого из 784 пикселов в изображениях из коллекции MNIST.
2. *Скрытый* слой, состоящий из 64 сигмоидных нейронов.
3. *Выходной* слой, состоящий из 10 нейронов типа softmax, по одному для каждого из 10 классов цифр.

Из них входной слой имеет самую простую организацию, поэтому начнем с него, а затем перейдем к скрытому и выходному слоям.

Нейроны во входном слое не выполняют никаких вычислений; они просто служат буфером для входных данных. Такая буферизация важна, потому что искусственные нейронные сети выполняют вычисления с матрицами, имеющими предопределенное число измерений и размеры. По меньшей мере одно из этих предварительно предопределенных измерений прямо соответствует форме входных данных.

ПОЛНОСВЯЗАННЫЙ СЛОЙ

Существует много видов скрытых слоев, но, как упоминалось в главе 4, чаще других используются *полносвязанные слои*, которые также иногда называют *плотными*. Полносвязанные слои присутствуют во многих архитектурах глубокого обучения, включая большинство моделей, которые мы рассмотрим в этой книге. Они имеют простую структуру: каждый нейрон в таком полносвязанном слое получает информацию от каждого из нейронов в предыдущем. Другими словами, полносвязанный слой имеет полный набор связей с предыдущим слоем!

Они могут быть не такими специализированными и не такими эффективными, как другие виды скрытых слоев, о которых мы расскажем в третьей части. Тем не менее полносвязанные слои в целом играют важную роль благодаря своей способности нелинейно рекомбинировать информацию, полученную от предыдущего слоя[1]. После обзора, представленного в разделе «Интерактивная среда TensorFlow» в конце главы 1, мы можем лучше оценить созданную модель глубокого обучения. Разбивая на отдельные слои сеть, изображенную на рис. 1.18 и 1.19, мы можем увидеть, как она организована.

1. *Входной слой* имеет два нейрона: один служит для хранения вертикальной координаты заданной точки в сетке справа, а другой — для хранения горизонтальной координаты.

2. *Скрытый слой состоит из восьми нейронов ReLU*. На схеме видно, что это полносвязанный слой, потому что каждый из восьми его нейронов подключен к обоим нейронам входного слоя, то есть получает от них информацию и всего имеет 16 (= 8 × 2) входящих соединений.

3. Второй *скрытый слой состоит из восьми нейронов ReLU*. И снова тип слоя легко определить, потому что каждый из его восьми нейронов получает информацию от каждого из восьми нейронов в предыдущем слое, то есть всего слой имеет 64 (= 8 × 8) входящих соединения. Обратите внимание на рис. 1.19, как нейроны в этом слое нелинейно рекомбинируют признаки краев, определяемые нейронами в первом скрытом слое, и получают более сложные признаки, такие как кривые и круги[2].

4. Третий *полносвязанный скрытый слой состоит из четырех нейронов ReLU* и имеет всего 32 (= 4 × 8) входящих соединения. Этот слой также нелинейно рекомбинирует информацию, получаемую из предыдущего скрытого слоя, и выявляет более сложные признаки, которые начинают выглядеть, как

---

[1] Это утверждение предполагает, что полносвязанный слой состоит из нейронов с нелинейной функцией активации, то есть таких, как sigmoid, tanh и ReLU, представленных в главе 6.

[2] Вернувшись в интерактивную среду plays.tensorflow.org, вы сможете детальнее исследовать эти признаки, наводя указатель мыши на соответствующие им нейроны.

имеющие прямое отношение к задаче бинарной классификации (оранжевый или синий), показанной в сетке справа на рис. 1.18.

5. Четвертый и последний *полносвязанный скрытый слой состоит из двух нейронов ReLU*. Он имеет всего 8 (= 2 × 4) входящих соединений с предыдущим слоем. Путем нелинейной рекомбинации нейроны в этом слое формируют настолько сложные признаки, что они визуально аппроксимируют общую границу, разделяющую синие и оранжевые точки на сетке справа.
6. *Выходной слой состоит из единственного сигмоидного нейрона*. Нейроны этого типа часто выбираются для подобных задач бинарной классификации. Как показано на рис. 6.9, сигмоидная функция возвращает активации в диапазоне от 0 до 1, что позволяет интерпретировать ее как оценку вероятности, что данный вход x является положительным случаем (в данном примере — синей точкой). Подобно скрытым слоям, *выходной слой тоже является полносвязанным*: его нейрон получает информацию от обоих нейронов последнего скрытого слоя, то есть всего имеет 2 (= 1 × 2) входящих соединения.

Таким образом, *все* слои в сети, демонстрируемой в интерактивной среде TensorFlow Playground, являются полносвязанными. Такую сеть можно назвать *полносвязанной* (или *плотной*), и с этими универсальными созданиями мы будем экспериментировать до конца второй части книги[1].

ПОЛНОСВЯЗАННАЯ СЕТЬ, ОПРЕДЕЛЯЮЩАЯ ХОТ-ДОГИ

Продолжим наше знакомство с полносвязанными сетями и вновь обратимся к простому бинарному классификатору, определяющему хот-доги, и математической нотации, использовавшейся для объяснения особенностей работы искусственных нейронов, из главы 6. Как показано на рис. 7.1, в этой главе наш классификатор больше не является единственным нейроном; теперь он реализован как полносвязанная сеть искусственных нейронов. Если говорить конкретнее, в эту сетевую архитектуру внесены следующие изменения:

- Для простоты количество входных нейронов сокращено до двух.
  - Первый входной нейрон, x_1, представляет объем кетчупа (скажем, в миллилитрах, сокращенно *мл*) в составе объекта, который анализируется

[1] Иногда полносвязанные, или плотные, сети называют *нейронными сетями прямого распространения* (feedforward neural networks) или многослойными перцептронами (MultiLayer Perceptron, MLP). Мы предпочитаем не использовать первый из этих двух терминов, потому что другие архитектуры, такие как сверточные нейронные сети (формально будут представлены в главе 10), тоже являются сетями прямого распространения (как и любые другие сети, не имеющие циклов). Второй термин мы предпочитаем не использовать, потому что, несмотря на название «многослойные перцептроны», эти сети не включают перцептроны, которые рассматривались в главе 6.

сетью. (Мы больше не используем перцептроны, поэтому не ограничены только бинарными входами.)

- Второй входной нейрон, x_2, представляет объем горчицы в миллилитрах.

- В сеть включено два полносвязанных скрытых слоя.
  - Первый скрытый слой имеет три нейрона ReLU.
  - Второй скрытый слой имеет два нейрона ReLU.
- Добавлен выходной нейрон, обозначенный как y. Наша задача относится к категории задач бинарной классификации, то есть как было отмечено в предыдущем разделе, этот нейрон должен иметь тип sigmoid. Так же как в примерах в главе 6, возвращая $y = 1$, сеть сообщает, что анализируемый объект является хот-догом, а возвращая $y = 0$ — что объект является чем-то другим.

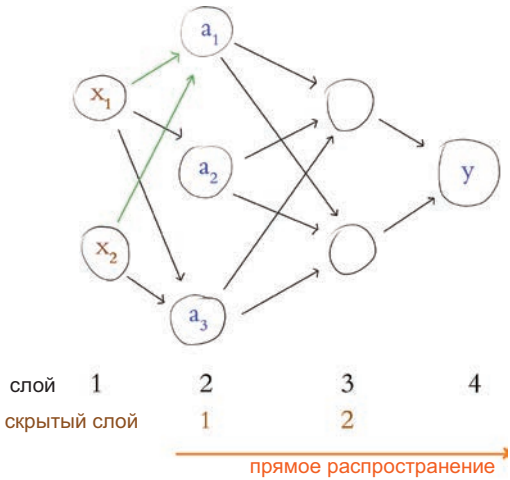


Рис. 7.1. Полносвязанная сеть искусственных нейронов, зеленым цветом выделены входы нейрона с меткой a_1

ПРЯМОЕ РАСПРОСТРАНЕНИЕ ЧЕРЕЗ ПЕРВЫЙ СКРЫТЫЙ СЛОЙ

Теперь, имея описание архитектуры сети для определения хот-догов, посмотрим, как она действует, сосредоточив внимание на нейроне с меткой a_1.[1] Этот конкретный нейрон, как и его соседи a_2 и a_3, получает информацию об объеме кетчупа и горчицы от x_1 и x_2 соответственно. Несмотря на то что a_1, a_2 и a_3 получают одни и те же данные, нейрон a_1 обрабатывает их по-своему, применяя свои уникальные параметры. Вспомнив рис. 6.7 с «самым важным уравнением в этой

[1] Для удобства идентификации нейронов мы решили использовать сокращенную запись. Более точные формальные обозначения для нейронных сетей вы найдете в приложении А.

книге» — $w \cdot x + b$, — можно понять причину этой уникальности. Развернув это уравнение для нейрона a_1, мы видим, что он имеет два входа из предыдущего слоя: x_1 и x_2. Он также имеет два веса: w_1 (определяет меру важности объема кетчупа x_1) и w_2 (определяет меру важности объема горчицы x_2). На основе этих пяти значений нейрон вычисляет z, то есть взвешенный вход этого нейрона:

$$\begin{aligned} z &= w \cdot x + b \\ z &= (w_1 x_1 + w_2 x_2) + b. \end{aligned} \tag{7.1}$$

Далее, опираясь на значение z, нейрон a_1 вычисляет активацию a для вывода. Поскольку нейрон a_1 является нейроном ReLU, вычисления выполняются в соответствии с уравнением на рис. 6.11:

$$a = \max(0; z). \tag{7.2}$$

Чтобы сделать вычисление выходного значения нейрона a_1 более конкретным, возьмем несколько чисел и подставим в формулы:

- $x_1 = 4.0$ мл кетчупа в данном объекте, который анализируется сетью;
- $x_2 = 3.0$ мл горчицы в этом же объекте;
- $w_1 = -0.5$;
- $w_2 = 1.5$;
- $b = -0.9$.

Теперь возьмем уравнение 7.1 и подставим в него конкретные значения:

$$\begin{aligned} z &= w \cdot x + b = \\ &= w_1 x_1 + w_1 x_2 + b = \\ &= -0.5 \times 4.0 + 1.5 \times 3.0 - 0.9 = \\ &= -2 + 4.5 - 0.9 = \\ &= 1.6. \end{aligned} \tag{7.3}$$

Наконец, чтобы найти a — выходную активацию нейрона a_1, — используем уравнение 7.2:

$$\begin{aligned} a &= \max(0, z) = \\ &= \max(0, 1.6) = \\ &= 1.6. \end{aligned} \tag{7.4}$$

Стрелка вправо, изображенная внизу на рис. 7.1, показывает порядок выполнения вычислений в слоях нейронной сети, от входного (значения x) до выходного (y), который называется *прямым распространением*. Только что мы подробно описали процесс прямого распространения через один нейрон

в первом скрытом слое нашей сети определения хот-догов. Прямое распространение через оставшиеся нейроны первого скрытого слоя, то есть вычисление значений a для нейронов a_2 и a_3, выполняется точно так же. Входы x_1 и x_2 идентичны для всех трех нейронов, но, несмотря на одинаковые получаемые значения, определяющие объем кетчупа и горчицы, каждый нейрон в первом скрытом слое выведет свою активацию a, отличную от других, потому что все они имеют свои уникальные параметры w_1, w_2 и b.

ПРЯМОЕ РАСПРОСТРАНЕНИЕ ЧЕРЕЗ ПОСЛЕДУЮЩИЕ СЛОИ

Прямое распространение через оставшиеся слои сети протекает практически так же, как через первый скрытый слой, но для ясности продолжим рассматривать пример вместе. На рис. 7.2 предполагается, что для всех нейронов в первом скрытом слое значения активации a уже вычислены. Активация, которую выводит нейрон a_1 (= 1.6), передается на один из трех входов нейрона a_4 (и, как показано на рисунке, та же активация $a = 1.6$ передается на один из трех входов нейрона a_5).

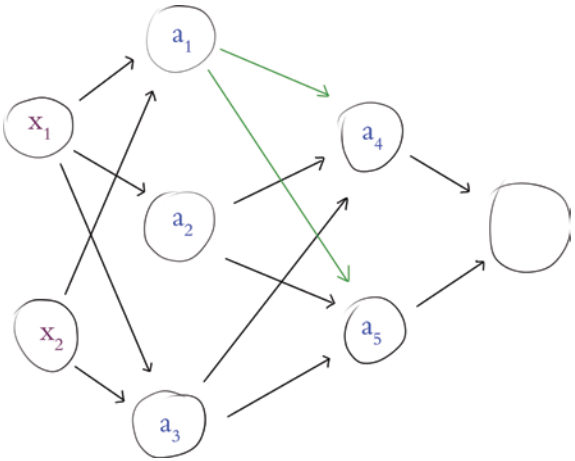


Рис. 7.2. Та же сеть определения хот-догов, что и на рис. 7.1, но теперь зеленым цветом выделена выходная активация нейрона a_1, которая передается на вход нейронов a_4 и a_5

В качестве примера прямого распространения через второй скрытый слой вычислим a для нейрона a_4. Снова используем самое важное уравнение $w \cdot x + b$, но для экономии места объединим его с функцией активации ReLU:

$$\begin{aligned} a &= \max(0; z) = \\ &= \max(0, (w \cdot x + b)) = \\ &= \max(0, (w_1x_1 + w_2x_2 + w_3x_3 + b)) \end{aligned} \tag{7.5}$$

Это уравнение очень похоже на уравнения 7.3 и 7.4, поэтому не будем тратить время на арифметику с вымышленными значениями. Единственное, что отличает прямое распространение через второй скрытый слой — входные данные (то есть x в уравнении $w \cdot x + b$) поступают не извне, а из первого скрытого слоя сети. То есть в уравнении 7.5:

- x_1 — это значение $a = 1.6$, полученное из нейрона a_1;
- x_2 — это значение a (чему бы оно ни было равно) из нейрона a_2;
- x_3 — аналогичная уникальная активация из нейрона a_3.

Соответственно, нейрон a_4 может нелинейно рекомбинировать информацию, предоставленную тремя нейронами из первого скрытого слоя. Нейрон a_5 тоже нелинейно рекомбинирует эту же информацию, но делает это по-своему: уникальные параметры w_1, w_2, w_3 и b в этом нейроне приведут к вычислению уникального значения активации.

Проиллюстрировав прямое распространение через все скрытые слои нашей сети определения хот-догов, перейдем к прямому распространению через выходной слой. Как показано на рис. 7.3, наш единственный нейрон в выходном слое получает входные данные от нейронов a_4 и a_5. Начнем с вычисления z для этого выходного нейрона. Вычисления выполняются по той же формуле из уравнения 7.1, которую мы использовали для вычисления z нейрона a_1, только значения (как обычно, выдуманные) переменных отличаются:

$$\begin{aligned} z &= w \cdot x + b = \\ &= w_1 x_1 + w_2 x_2 + b = \\ &= 1.0 \times 2.5 + 0.5 \times 2.0 - 5.5 = \\ &= 3.5 - 5.5 = \\ &= -2.0 \end{aligned} \tag{7.6}$$

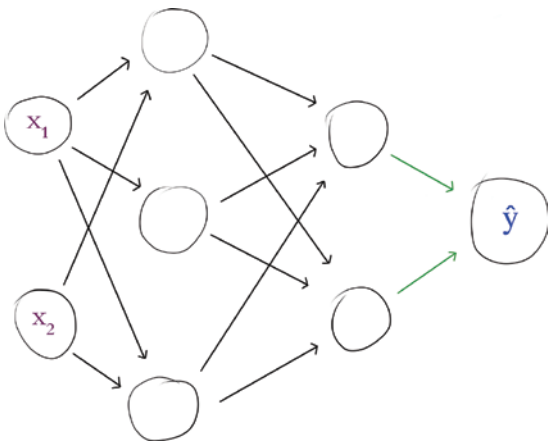


Рис. 7.3. Все та же сеть определения хот-догов, теперь зеленым цветом выделены активации, которые передаются на вход выходного нейрона $\hat{y}$

В роли выходного используется сигмоидный нейрон, поэтому его активация вычисляется передачей значения z в сигмоидную функцию на рис. 6.9:

$$\begin{aligned} a &= \sigma(z) \\ &= \frac{1}{1+e^{-z}} = \\ &= \frac{1}{1+e^{-(-2.0)}} = \\ &\approx 0.1192 \end{aligned} \tag{7.7}$$

Мы довольно ленивы, поэтому окончательный результат в последней строке этого уравнения вычислили не вручную, а использовали блокнот Jupyter *sigmoid_function.ipynb*, который создали в главе 6. Выполнив в нем строку `sigmoid(-2.0)`, наш компьютер взял на себя всю тяжелую работу и любезно сообщил, что значение a составляет примерно `0.1192`.

Активация a, вычисляемая сигмоидным нейроном в выходном слое, — это особый случай, потому что он является конечным результатом всей нейронной сети, определяющей хот-доги. Чтобы подчеркнуть его особенность, мы использовали особое обозначение: $\hat{y}$. Имя этой переменной состоит из буквы y и «крышки» над ней, чтобы она могла держать голову в тепле, поэтому мы называем ее «игрек-крышка». Значение переменной $\hat{y}$ является предположением сети о том, является ли данный объект хот-догом, выраженным на языке вероятностей. Опираясь на входные данные x_1 и x_2, то есть 4.0 мл кетчупа и 3.0 мл горчицы, сеть оценила вероятность, что объект с этими конкретными объемами приправ является хот-догом, как 11.92%[1]. Если объект, информация о котором попала в сеть, действительно был хот-догом ($y = 1$), тогда прогноз $\hat{y} = 0.1192$ довольно далек от истины. С другой стороны, если объект не является хот-догом ($y = 0$), тогда прогноз $\hat{y}$ можно считать довольно точным. Мы формализуем оценку предсказаний $\hat{y}$ в главе 8, но в целом, чем ближе $\hat{y}$ к истинному значению y, тем лучше.

СЛОЙ SOFTMAX ДЛЯ СЕТИ КЛАССИФИКАЦИИ ФАСТФУДА

Как было показано выше в этой главе, сигмоидный нейрон удобно использовать в роли выходного нейрона в сети, различающей два класса, таких как синие и оранжевые точки, или хот-дог и другие объекты, не являющиеся хот-догами. Однако часто бывает нужно различать более двух классов. Например, коллекция MNIST содержит изображения 10 цифр, поэтому наша неглубокая сеть из главы 6 должна была выводить 10 вероятностей — по одной для каждой цифры.

[1] Не говорите, что мы не предупреждали, что это глупый пример! Если повезет, его нелепость поможет вам крепче запомнить его.

В задачах классификации с несколькими классами предпочтительнее в роли выходного использовать слой softmax. Softmax — это функция активации, которую мы выбрали для выходного слоя в блокноте Jupyter *shallow_net_in_keras.ipynb* (пример 5.6), но тогда мы предложили вам не беспокоиться об этой детали. Теперь, пару глав спустя, пришло время разгадать тайну softmax.

На рис. 7.4 представлена новая архитектура, основанная на бинарном классификаторе хот-догов. Она имеет ту же организацию — вплоть до входов, через которые вводятся объемы кетчупа и горчицы, только вместо одного выходного нейрона у нас их теперь три. Этот мультиклассовый выходной слой все еще является полносвязанным, поэтому каждый из трех его нейронов получает информацию от обоих нейронов в последнем скрытом слое. Продолжая аналогию с фастфудом, скажем теперь:

- y_1 представляет хот-доги;
- y_2 представляет гамбургеры;
- y_3 представляет пиццу.

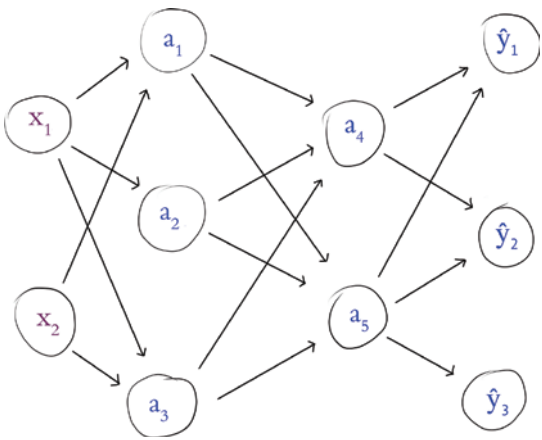


Рис. 7.4. Сеть классификации фастфуда с тремя нейронами softmax в выходном слое

Обратите внимание, что в этой конфигурации не может быть альтернатив хот-догам, гамбургерам или пицце. Предполагается, что все объекты, информация о которых передается в сеть, принадлежат к одному и только к одному из этих трех классов фастфуда.

Функция sigmoid используется только для задач бинарной классификации, поэтому в выходных нейронах на рис. 7.4 применяется функция активации softmax. Используем код из блокнота Jupyter *softmax_demo.ipynb* и выясним, как работает эта функция активации. Единственная необходимая зависимость — функция `exp`, которая вычисляет натуральную экспоненту любого переданного ей значения. То есть если передать ей некоторое значение x, выполнив команду `exp(x)`, она вернет e^x. Смысл этого возведения в степень станет ясен, когда мы

исследуем следующий пример. Импортируем функцию `exp` в блокнот с помощью инструкции `from math import exp`.

Для большей конкретики предположим, что мы передали в сеть на рис. 7.4 информацию о куске пиццы. Он содержит незначительные количества кетчупа и горчицы, поэтому значения x_1 и x_2 близки к нулю. Далее эта информация распространяется вперед по сети до ее выходного слоя. Основываясь на информации, которую три нейрона получают из последнего скрытого слоя, они по отдельности используют нашу формулу $w \cdot x + b$ и вычисляют три уникальных (и вымышленных, для целей этого примера) значения z:

- для нейрона $\hat{y}_1$, который представляет хот-доги, z получает значение -1.0;
- для нейрона $\hat{y}_2$, который представляет гамбургер, z получает значение 1.0;
- для нейрона $\hat{y}_1$, который представляет пиццу, z получает значение 5.0.

Эти значения говорят о том, что по оценкам сети данный объект наиболее вероятно является пиццей, а наименее вероятно — хот-догом. Однако по значениям z трудно сказать, насколько более вероятно, что данный объект является пиццей, по сравнению с двумя другими классами. Вот тут на помощь нам приходит функция softmax.

Импортировав зависимость, создаем список с именем `z` для хранения трех значений z:

```
z = [-1.0, 1.0, 5.0]
```

Применение функции softmax к этому списку выполняется в три этапа. Сначала вычисляется экспонента для каждого из значений z. То есть:

- `exp(z[0])` дает в результате `0.3679` для хот-дога[1];
- `exp(z[1])` дает `2.718` для гамбургера;
- `exp(z[2])` дает намного, намного большее значение (все-таки экспонента!) `148.4` для пиццы.

На втором шаге функция softmax вычисляет сумму экспонент:

```
total = exp(z[0]) + exp(z[1]) + exp(z[2])
```

И затем, на третьем шаге, переменная `total` используется для вычисления пропорциональной доли для каждого из трех классов:

- `exp(z[0])/total` дает значение $\hat{y}_1$, равное `0.002428`, указывающее, что по оценкам сети вероятность принадлежности объекта к классу хот-догов составляет ~0.2%;

[1] Не забывайте, что счет индексов в Python начинается с нуля, поэтому `z[0]` соответствует значению z нейрона $\hat{y}_1$.

- `exp(z[1])/total` дает значение $\hat{y}_2$, равное `0.01794`, — вероятность принадлежности объекта к классу гамбургеров составляет ~1.8%;
- `exp(z[2])/total` дает значение $\hat{y}_3$, равное `0.9796`, — вероятность принадлежности объекта к классу пиццы составляет ~98.0%.

Опираясь на эту арифметику, легко понять значение имени «softmax»: функция возвращает z с наибольшим значением (*max*), но делает это мягко (*softly*). То есть сеть не сообщает, что объект со стопроцентной вероятностью является пиццей и с вероятностью 0 процентов принадлежит двум другим классам фастфуда (такой результат дала бы функция *hardmax*), а выражает сомнение и возвращает вероятность принадлежности объекта к каждому из трех классов, предлагая нам самим принять решение о верности прогноза[1].

Использование функции единственного нейрона softmax — это особый случай, математически эквивалентный использованию сигмоидного нейрона.

ПОВТОРНЫЙ ОБЗОР НЕГЛУБОКОЙ СЕТИ

Познакомившись с полносвязанными сетями в этой главе, вернемся к блокноту *shallow_net_in_keras.ipynb* и рассмотрим еще раз эту модель. В листинге 5.6 показаны три строки кода, использующие библиотеку Keras, которые конструируют неглубокую нейронную сеть для классификации цифр из коллекции MNIST. Как рассказывалось в главе 5, эти три строки кода создают экземпляр модели и добавляют в нее слои искусственных нейронов. Вызвав метод `summary()` модели, можно увидеть таблицу со сводной информацией о ней, как показано на рис. 7.5. Таблица имеет три столбца:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

Рис. 7.5. Сводная информация об объекте модели из блокнота shallow_net_in_keras.ipynb

[1] Пороговые значения достоверности могут меняться в зависимости от конкретного приложения, но обычно выбирается класс с наибольшей вероятностью. Этот класс можно идентифицировать, например, с помощью функции `argmax()` в Python, которая возвращает индекс (то есть метку класса) с наибольшим значением.

- `Layer (type)`: название и тип каждого слоя;
- `Output Shape`: размерность слоя;
- `Param #`: количество параметров (весов w и смещений b) слоя.

Входной слой не выполняет никаких вычислений и не имеет своих параметров, поэтому информация о нем не отображается напрямую. Поэтому первая строка в таблице соответствует первому скрытому слою сети. В таблице указано, что этот слой:

- называется `dense_1`; это имя по умолчанию, потому что мы задали его явно;
- имеет тип `Dense`, как мы задали в листинге 5.6;
- состоит из 64 нейронов, что мы также задали в листинге 5.6;
- имеет 50 240 параметров, в том числе:
  - 50 176 весов, по числу связей 64 нейронов в этом полносвязанном слое с каждым из 784 нейронов во входном слое (64 × 784);
  - 64 смещения, по одному для каждого нейрона в слое;
  - в сумме получается 50 240 параметров:

$$n_{\text{параметров}} = n_w + n_b = 50\,176 + 64 = 50\,240.$$

Вторая строка в таблице на рис. 7.5 соответствует выходному слою модели. Как видите, этот слой:

- называется `dense_2`;
- имеет тип `Dense`, как мы задали в примере;
- состоит из 10 нейронов, что мы также задали в примере;
- имеет 650 параметров, в том числе:
  - 640 весов, по числу связей 10 нейронов в этом слое с каждым из 64 нейронов в предшествующем скрытом слое (64 × 10);
  - 10 смещений, по одному для каждого нейрона в выходном слое.

Сложив число параметров в каждом слое, получаем общее число параметров в сети, которое показано в строке `Total params` на рис. 7.5:

$$\begin{aligned} n_{total} &= n_1 + n_2 = \\ &= 50\,240 + 650 = \\ &= 50\,890. \end{aligned} \tag{7.8}$$

Все эти 50 890 параметров являются обучаемыми (строка `Trainable params`), потому что они могут настраиваться во время обучения модели, в вызове `model.`

`fit()` в блокноте *shallow_net_in_keras.ipynb*. Это нормальное явление, но, как будет показано в третьей части, иногда бывает полезно зафиксировать некоторые параметры, превратив их в необучаемые.

ИТОГИ

В этой главе мы узнали, как искусственные нейроны объединяются в сеть для аппроксимации выходных данных y по входным значениям x. В оставшихся главах второй части мы подробно рассмотрим, как сеть учится улучшать свои аппроксимации для y, используя обучающие данные для настройки параметров своих искусственных нейронов. Одновременно мы познакомимся с новыми приемами проектирования и обучения искусственных нейронных сетей, позволяющими включать дополнительные скрытые слои и формировать модели глубокого обучения высокого качества.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым.

- параметры:
  - вес w;
  - смещение b;
- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
- входной слой;
- скрытый слой;
- выходной слой;
- типы слоев:
  - плотный (полносвязанный);
  - softmax;
- прямое распространение.

8

ОБУЧЕНИЕ ГЛУБОКИХ СЕТЕЙ

В предыдущих главах мы подробно описали искусственные нейроны и процесс прямого распространения информации через нейронную сеть до получения прогноза, например: является ли данный образец фастфуда хот-догом, сочным гамбургером или хорошим куском пиццы. В этих кулинарных примерах из глав 6 и 7 мы сами выдумали значения для параметров нейронов — весов и смещений. Однако в реальных приложениях эти параметры обычно не берутся с потолка, а вычисляются в ходе обучения сети.

В этой главе вы познакомитесь с двумя методами обучения — *градиентным спуском* и *обратным распространением*, которые применяются для вычисления параметров искусственной нейронной сети. Как обычно, мы представим не только теоретическое обоснование методов, но и их практическую реализацию. Кульминацией этой главы станет применение этих методов для построения нейронной сети с несколькими скрытыми слоями.

ФУНКЦИИ СТОИМОСТИ

В главе 7 мы узнали, что при прямом распространении входных значений через искусственную нейронную сеть на ее выходе получается результат, который обозначается как $\hat{y}$. Идеально откалиброванная сеть могла бы вывести значения $\hat{y}$, которые точно равны истинным меткам y. В нашем бинарном классификаторе хот-догов, например (рис. 7.3), $y = 1$ указывает, что объект, информация о котором передана в сеть, является хот-догом, а $y = 0$ — что он является чем-то другим. Поэтому при передаче в сеть информации о хот-доге в идеале она вывела бы $\hat{y} = 1$.

На практике «золотой стандарт» $\hat{y} = y$ достигается не всегда и может оказаться чрезмерно строгим определением «правильного» $\hat{y}$. Поэтому часто для $y = 1$ нам достаточно видеть результат $\hat{y}$, равный, например, 0.9997, означающий достаточно высокую уверенность сети в том, что объект является хот-догом.

Значение $\hat{y} = 0.9$ можно считать приемлемым, $\hat{y} = 0.6$ — разочаровывающим, а $\hat{y} = 0.1192$ (как в уравнении 7.7) — ужасным.

Для количественного выражения спектра качественных оценок результатов от «замечательных» до «ужасных» в алгоритмы машинного обучения часто включают *функции стоимости* (также известные как *функции потерь*). В этой книге мы изучим две такие функции: квадратичную функцию стоимости и перекрестную энтропию. Рассмотрим их по очереди.

КВАДРАТИЧНАЯ ФУНКЦИЯ СТОИМОСТИ

Квадратичная функция стоимости — одна из простейших с вычислительной точки зрения. Также ее называют *среднеквадратичной ошибкой*, что объясняется следующей формулой:

$$C = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2. \tag{8.1}$$

Для каждого i-го экземпляра вычисляется разность (*ошибка*) между истинной меткой y_i и оценкой сети $\hat{y}_i$, и затем эта разность возводится в *квадрат*, потому что:

1. Возведение в квадрат гарантирует, что, если y больше, чем $\hat{y}$, или наоборот, разность всегда будет иметь положительное значение.
2. Возведение в квадрат увеличивает штраф за большую разность между y и $\hat{y}$.

Получив *квадратичную ошибку* для каждого i-го экземпляра, как $(y_i - \hat{y}_i)^2$, мы можем вычислить *среднюю стоимость* C для всех n экземпляров:

1. Сложить стоимости для всех экземпляров: $\sum_{i=1}^{n}$.
2. Разделить сумму на число экземпляров, умножив ее на $\frac{1}{n}$.

Открыв блокнот Jupyter *quadratic_cost.ipynb* из репозитория GitHub с примерами для книги, вы сможете поиграть с уравнением 8.1. В начале блокнота определяется функция вычисления квадратичной ошибки для i-го экземпляра:

```
def squared_error(y, yhat):
    return (y - yhat)**2
```

Передав в функцию истинное значение `y`, равное `1`, и идеальное значение `yhat`, равное `1`, с помощью команды `squared_error(1, 1)`, можно увидеть, что эта идеальная (и желаемая) оценка связана со стоимостью `0`. Незначительные отклонения от идеального значения, такие как `yhat = 0.9997`, соответствуют чрезвычайно малым стоимостям: `9.0e-08`[1]. По мере увеличения разности между `y` и `yhat` наблюдается ожидаемое экспоненциальное увеличение стоимости:

[1] `9.0e-08` эквивалентно числу 9.0×10^{-8}.

для истинного значения `y`, равного `1`, и уменьшающегося значения **yhat** с `0.9` до `0.6` и до **0.1192** стоимость быстро увеличивается с `0.01` до `0.16` и затем до `0.78`. Наконец, ради эксперимента попробуйте в блокноте вычислить стоимость для истинного значения `y`, равного `0`, и значения `yhat`, равного `0.1192`; вы увидите, что в этом случае она получится небольшой: `0.0142`.

НАСЫЩЕННЫЕ НЕЙРОНЫ

Квадратичная стоимость является простым представителем функций потерь, но она имеет существенный недостаток. Взгляните на рис. 8.1, на котором повторно воспроизводится функция активации tanh, представленная ранее на рис. 6.10. Проблема, изображенная на рисунке, называется *насыщением нейронов* и характерна для всех функций активации, но далее в качестве образца мы будем использовать только функцию активации tanh. Нейрон считается насыщенным, когда комбинация его входов и параметров (взаимодействующих согласно «самому важному уравнению» $z = w \cdot x + b$, изображенному на рис. 6.10) дает экстремальные значения z в областях, обведенных овалом на рис. 8.1. В этих областях значительные изменения z (путем корректировки параметров нейрона w и b) вызывают незначительные изменения в активации нейрона a.[1]

Используя методы, которые мы рассмотрим далее в этой главе — градиентный спуск и обратное распространение, — нейронная сеть способна научиться аппроксимировать y путем корректировки параметров w и b всех своих нейронов. В насыщенном нейроне, когда изменения w и b дают лишь незначительные изменения в a, обучение существенно замедляется: если корректировки w и b не оказывают заметного влияния на активацию данного нейрона a, то они не способны сколько-нибудь заметно влиять (через прямое распространение) на результат $\hat{y}$ сети — ее оценку y.

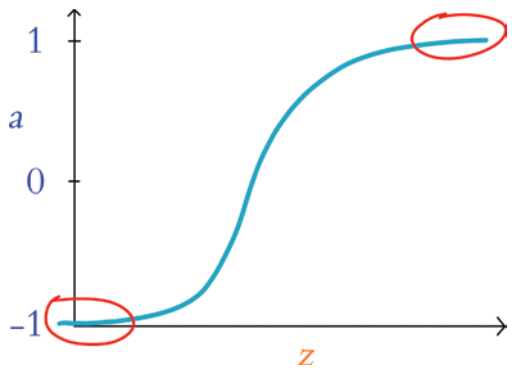


Рис. 8.1. График функции активации tanh, взятый из рис. 6.10. Обратите внимание на области низких и высоких значений `z`, в которых возникает эффект насыщения нейронов

[1] Как рассказывалось в главе 6, $a = \sigma(z)$, где σ — некоторая функция активации, в данном примере функция tanh.

ПЕРЕКРЕСТНАЯ ЭНТРОПИЯ

Одним из способов[1] уменьшить отрицательное влияние насыщенных нейронов на скорость обучения является использование *перекрестной энтропии* вместо функции квадратичной стоимости. Эта альтернативная функция потерь обеспечивает эффективное обучение в любой точке на кривой функции активации на рис. 8.1. По этой причине она пользуется большей популярностью в роли функции стоимости, и именно ее мы будем использовать в большинстве примеров[2].

Конкретная формула вычисления перекрестной энтропии не имеет большого значения для наших целей, тем не менее приведем ее для полноты картины:

$$C = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln \hat{y}_i + (1-y_i)\ln\left(1-\hat{y}_i\right)\right]. \tag{8.2}$$

Вот несколько важных замечаний, касающихся этой формулы:

- так же как в квадратичной функции стоимости, увеличение разности между $\hat{y}$ и y влечет увеличение стоимости;
- по аналогии с квадратом в квадратичной функции стоимости натуральный логарифм *ln* в функции перекрестной энтропии влечет экспоненциальное увеличение стоимости при линейном увеличении разности между $\hat{y}$ и y;
- функция перекрестной энтропии структурирована так, что чем больше разность между $\hat{y}$ и y, *тем выше скорость обучения нейрона*[3].

Чтобы проще было понять, как высокая стоимость способствует ускорению обучения нейронной сети, использующей функцию стоимости на основе перекрестной энтропии, приведем аналогию, которая не имеет ничего общего ни с кем из уважаемых авторов книги: допустим, вы находитесь на коктейльной

---

[1] Другие методы уменьшения отрицательного влияния насыщенных нейронов на сеть описываются в главе 9.

[2] Перекрестная энтропия хорошо подходит для нейронных сетей, решающих задачи классификации, каковых большинство в этой книге. Для задач регрессии (описываются в главе 9) лучше подходит квадратичная стоимость.

[3] Чтобы понять, как функция перекрестной энтропии из уравнения 8.2 способствует увеличению скорости обучения нейрона с большей стоимостью, нужно затронуть тему вычисления частной производной. (Поскольку мы стремимся свести к минимуму использование сложного математического аппарата в этой книге, мы вынесли объяснение в сноску.) Основой двух вычислительных методов обучения нейронных сетей — градиентного спуска и обратного распространения — является сравнение скорости изменения стоимости C относительно параметров нейрона, таких как вес w. Используя обозначение частной производной, эти относительные скорости изменения можно представить как $\frac{\partial C}{\partial w}$. Функция перекрестной энтропии преднамеренно структурирована так, что при вычислении ее производной $\frac{\partial C}{\partial w}$ соотносится с $(\hat{y} - y)$. То есть чем больше разность между идеальным и расчетным выходами нейрона y и $\hat{y}$, тем выше скорость изменения стоимости C относительно веса w.

вечеринке и ведете беседу с группой людей, которых впервые встретили этим вечером. Крепкий мартини уже ударил вам в голову, и вы вводите во вполне обычный ответ некоторую шутку на грани дозволенного. Ваши собеседники немедленно реагируют с видимым отвращением. Такая реакция окружающих, четко указывающая, что шутка не достигла цели, чертовски быстро научит вас соблюдать нормы приличия. Крайне маловероятно, что вы рискнете повторить эту шутку в ближайшее время.

Во всяком случае, этого вполне достаточно с точки зрения социального этикета. Последнее, что важно отметить в отношении перекрестной энтропии: наличие $\hat{y}$ в уравнении 8.2 говорит о том, что оно применимо только к выходному слою. Однако, как рассказывалось в главе 7 (в частности, в обсуждении рис. 7.3), $\hat{y}$ является частным случаем a: на самом деле это то же самое значение a, просто оно вычисляется нейронами в выходном слое сети. Учитывая это, уравнение 8.2 можно выразить через a_i, заменив им $\hat{y}_i$, и тем самым обобщить уравнение для любого слоя сети:

$$C = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln a_i + (1-y_i)\ln(1-a_i)\right]. \tag{8.3}$$

Чтобы закрепить всю эту теоретическую болтовню о перекрестной энтропии, давайте поэкспериментируем с блокнотом Jupyter *cross_entropy_cost.ipynb*. Блокнот подключает только одну зависимость: функцию `log` из пакета *NumPy*, которая вычисляет натуральный логарифм *ln*, присутствующий в уравнении 8.3. Эта зависимость загружается инструкцией `from numpy import log`.

Далее определяется функция вычисления стоимости на основе перекрестной энтропии для *i*-го экземпляра:

```
def cross_entropy(y, a):
    return -1*(y*log(a) + (1-y)*log(1-a))
```

Таблица 8.1. Стоимость на основе перекрестной энтропии для выбранных входных примеров

y	*a*	*C*
1	0.9997	0.0003
1	0.9	0.1
1	0.6	0.5
1	0.1192	2.1
0	0.1192	0.1269
1	1–0.1192	0.1269

Подставляя в функцию `cross_entropy()` те же значения, что и в `squared_error()` выше в этой главе, мы наблюдаем сопоставимое поведение. Как показано в табл. 8.1, удерживая y на уровне `1` и постепенно уменьшая a от почти идеальной

оценки `0.9997` до `0.1192`, мы получаем экспоненциальный рост стоимости на основе перекрестной энтропии. Далее таблица иллюстрирует, что — опять же в соответствии с поведением ее квадратичной родственницы — стоимость на основе перекрестной энтропии получается низкой для a, равной `0.1192`, если y фактически равно `0`. Эти результаты подтверждают, что главное отличие перекрестной энтропии от квадратичной стоимости — не конкретное значение стоимости, а скорость, с которой протекает обучение нейронной сети, особенно при наличии насыщенных нейронов.

ОПТИМИЗАЦИЯ: ОБУЧЕНИЕ МЕТОДОМ МИНИМИЗАЦИИ СТОИМОСТИ

Функции стоимости дают нам количественную оценку того, насколько ошибочен прогноз модели относительно идеального значения y. То есть мы получаем метрику, которую можно использовать для уменьшения ошибки сети.

Как уже упоминалось в этой главе, основной подход к минимизации стоимости в задачах глубокого обучения основан на объединении двух методов: градиентного спуска и обратного распространения. Эти методы являются *оптимизаторами* и обеспечивают *обучение* сети. Обучение заключается в корректировке параметров модели для постепенного приближения оценочного значения $\hat{y}$ к целевому значению y, и, соответственно, снижения стоимости. Сначала мы рассмотрим градиентный спуск, а затем перейдем к обратному распространению.

ГРАДИЕНТНЫЙ СПУСК

Градиентный спуск — это удобный и эффективный инструмент для настройки параметров модели с целью минимизации стоимости, особенно при наличии большого объема обучающих данных. Он широко используется не только в глубоком обучении, но и в машинном обучении в целом.

Для иллюстрации градиентного спуска (рис. 8.2) мы использовали трилобита. Горизонтальная ось в каждом кадре соответствует некоторому параметру, который мы обозначили как p. В искусственной нейронной сети этим параметром будет либо вес нейрона w, либо смещение b. В верхнем кадре трилобит находится на возвышении. Его цель — *спуститься* по склону, чтобы найти точку с минимальной стоимостью C. Но есть одна проблема: трилобит слеп! Он не видит, имеются ли более глубокие участки где-то вдали, и может использовать только свою трость для исследования наклона местности в непосредственной близости.

Оранжевая пунктирная линия на рис. 8.2 показывает оценку наклона склона слепым трилобитом в точке своего местонахождения. Согласно этой оценке, если трилобит сделает шаг влево (то есть переместится чуть ниже по склону,

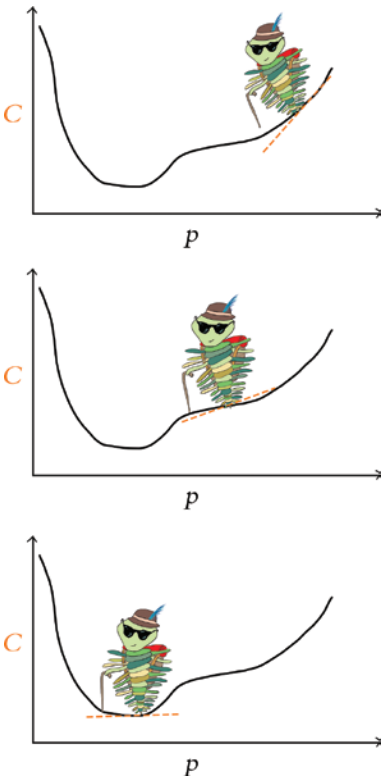


Рис. 8.2. Трилобит использует метод градиентного спуска, чтобы найти значение параметра *p*, для которого стоимость *C* окажется минимальной

к более низкому значению *p*), он переместится в точку с меньшей стоимостью. С другой стороны, сделав шаг вправо (чуть выше по склону, к более *высокому* значению *p*), он переместится в точку с *большей* стоимостью. Следуя своему желанию спуститься вниз по склону, трилобит решает сделать шаг влево.

На среднем кадре трилобит уже сделал несколько шагов влево. И снова мы видим, как он, оценив наклон оранжевой линии, приходит к выводу, что шаг влево приведет его в точку с более низкой стоимостью, и он делает этот шаг. В нижнем кадре трилобиту удалось добраться до точки — значения параметра *p* — с минимальной стоимостью. Если теперь он сделает шаг влево или вправо, стоимость вырастет, поэтому он, довольный достигнутым результатом, остается на месте.

На практике модель глубокого обучения имеет не единственный параметр. Нередко число параметров в сетях глубокого обучения исчисляется миллионами, а в некоторых промышленных приложениях и миллиардами. Даже наша неглубокая сеть в блокноте *shallow_net_in_keras.ipynb* — одна из самых маленьких моделей, которые мы создадим в этой книге, — имеет 50 890 параметров (см. рис. 7.5).

Человеческий разум не способен представить пространство с миллиардом измерений, но модель с двумя параметрами, показанная на рис. 8.3, позволяет

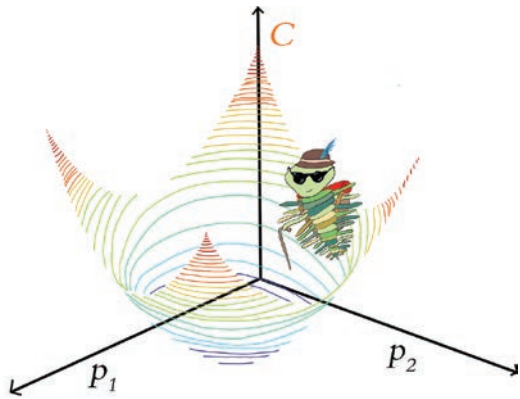


Рис. 8.3. Трилобит исследует два параметра — p_1 и p_2, — пытаясь минимизировать стоимость методом градиентного спуска. Проводя аналогию с приключениями в горах, параметры p_1 и p_2 можно рассматривать как широту и долготу, а стоимость — как высоту

получить представление о том, как можно использовать градиентный спуск для минимизации стоимости по нескольким параметрам одновременно. Независимо от числа обучаемых параметров в модели, градиентный спуск итеративно оценивает наклон[1] и определяет, на какую величину следует скорректировать каждый параметр, чтобы выбрать направление наибольшего снижения стоимости. С двумя параметрами, как, например, на рис. 8.3, эту процедуру можно сравнить с походом слепого трилобита через горы, где:

- один параметр, например p_1, представляет широту;
- второй параметр, p_2, представляет долготу;
- а стоимость представляет высоту — чем меньше высота, тем лучше!

Трилобит попадает в случайное место в горах. Он начинает ощупывать окружающее его пространство своей тростью, пытаясь определить направление, чтобы шагнуть и максимально уменьшить высоту своего местоположения. Затем он делает этот единственный шаг. Повторяя этот процесс много раз, трилобит может в конечном итоге оказаться в координатах, соответствующих минимально возможной высоте (минимальной стоимости), после чего сюрреалистическое горное приключение трилобита завершится.

СКОРОСТЬ ОБУЧЕНИЯ

Для простоты вернемся к слепому трилобиту на рис. 8.4, перемещающемуся в двумерном мире с единственным параметром. Теперь представим, что у нас есть лучевая пушка, способная уменьшать и увеличивать трилобитов. На среднем кадре мы использовали пушку, чтобы уменьшить трилобита. Такой трилобит делает очень короткие шаги, и поэтому нашему маленькому бесстрашному путешественнику потребуется много времени, чтобы найти путь в легендарную

[1] Вычисляя частные производные.

долину с минимальной стоимостью. А теперь взгляните на нижний кадр, где мы использовали пушку и увеличили трилобита. Ситуация стала еще хуже! Теперь трилобит делает настолько большие шаги, что легко может перешагнуть через долину минимальной стоимости и никогда не сможет найти ее.

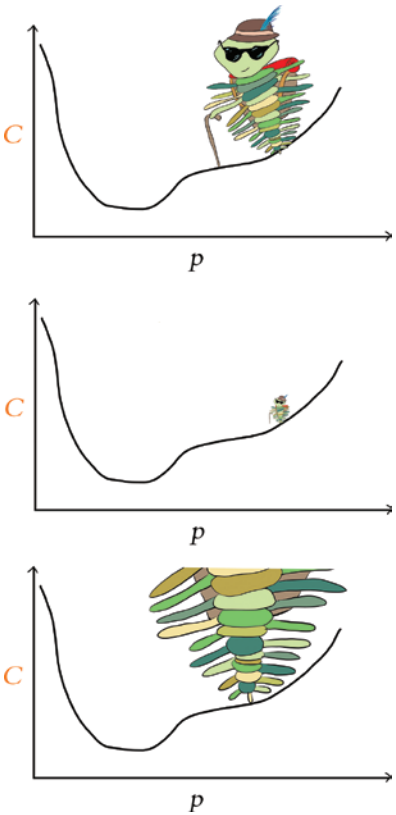


Рис. 8.4. Скорость обучения (η) методом градиентного спуска определяется размером трилобита. На среднем кадре изображен маленький трилобит с низкой скоростью обучения, а на нижнем — большой, с высокой скоростью обучения

В терминологии градиентного спуска размер шага называется *скоростью обучения* и обозначается греческой буквой η (эта). Скорость обучения — это первый из *гиперпараметров* моделей, которые мы рассмотрим в этой книге. В машинном обучении, включая глубокое обучение, гиперпараметрами называют аспекты модели, которые настраиваются перед началом обучения. Поэтому гиперпараметры, такие как η, настраиваются перед обучением, а w и b — во время обучения.

К правильным значениям гиперпараметров модели глубокого обучения часто приходят методом проб и ошибок. Настройки скорости обучения η — как в сказке «Три медведя»: слишком маленькая и слишком большая — никуда не годятся, а между ними находится оптимальное значение. В частности, как показано на рис. 8.4, если скорость η слишком мала, потребуется много-много итераций градиентного спуска (читай: слишком много времени), чтобы достичь

области минимальной стоимости. С другой стороны, если выбрать слишком большое значение, можно никогда не достичь точки с минимальной стоимостью: алгоритм градиентного спуска будет действовать хаотично, перепрыгивая через значения параметров, соответствующие минимальной стоимости.

Далее, в главе 9, нас поджидает хитрый трюк, который избавит от необходимости вручную подбирать гиперпараметр для этой нейронной сети. А пока познакомьтесь с нашими практическими рекомендациями по этому вопросу:

- На начальном этапе используйте скорость обучения около 0.01 или 0.001.
- Если модель проявит способность к обучению (то есть если стоимость будет последовательно уменьшаться от эпохи к эпохе), но оно происходит очень медленно (в каждую эпоху стоимость снижается на небольшую величину), то увеличьте скорость обучения на порядок (например, от 0.01 до 0.1). Если стоимость начинает беспорядочно скакать вверх-вниз от эпохи к эпохе, значит, вы зашли слишком далеко и нужно уменьшить скорость обучения.
- Напротив, если модель не проявляет способности к обучению, это может служить признаком слишком высокой скорости обучения. Попробуйте уменьшить ее на порядок (например, с 0.001 до 0.0001), пока стоимость не будет последовательно уменьшаться от эпохи к эпохе. Чтобы получить представление о нестабильном поведении модели, когда скорость обучения слишком высока, вернитесь к примеру интерактивной среды TensorFlow, изображенной на рис. 1.18, и выберите большое значение в раскрывающемся списке Learning rate (Скорость обучения).

РАЗМЕР ПАКЕТА И СТОХАСТИЧЕСКИЙ ГРАДИЕНТНЫЙ СПУСК

Знакомясь с градиентным спуском, мы упомянули, что он эффективен в задачах машинного обучения с использованием больших наборов обучающих данных. Строго говоря, мы погрешили против истины. На самом деле при наличии большого объема обучающих данных обычный градиентный спуск вообще не сработает из-за невозможности поместить все данные в оперативную память (ОЗУ) компьютера.

Память — не единственная потенциальная проблема; недостаточность вычислительной мощности тоже может превратиться в головную боль. В память компьютера можно втиснуть относительно большой набор данных, но если попытаться обучить нейронную сеть с миллионами параметров на этих данных, классический градиентный спуск окажется крайне *не*эффективным из-за вычислительной сложности обработки большого объема многомерных данных.

К счастью, проблема ограниченной памяти и вычислительной мощности имеет решение: *стохастический* вариант градиентного спуска (Stochastic Gradient Descent, SGD). При использовании этого варианта обучающие данные раз-

биваются на *пакеты* — небольшие подмножества полного обучающего набора данных, чтобы сделать градиентный спуск управляемым и продуктивным.

Об этом не говорилось явно, но мы уже использовали стохастический градиентный спуск в главе 5, когда обучали модель в блокноте *shallow_net_in_keras.ipynb* и устанавливали значение `SGD` для оптимизатора на шаге `model.compile()`. В следующей строке кода, где вызывается метод `model.fit()`, мы присвоили значение `128` переменной `batch_size`, которая определяет размер пакетов — количество обучающих образцов для данной итерации SGD. Как и скорость обучения η, представленная выше в этой главе, *размер пакета* также является гиперпараметром модели.

Рассмотрим пример с конкретными числами, чтобы сделать понятия пакета и стохастического градиентного спуска более осязаемыми. В наборе данных MNIST имеется 60 000 обучающих изображений. Если ограничить размер пакета 128 изображениями, мы получаем $\lceil 468.75 \rceil = 469$[1,2] пакетов на эпоху градиентного спуска:

$$\begin{aligned} \text{число пакетов} &= \left\lceil \frac{\text{размер обучающего набора}}{\text{размер пакета}} \right\rceil = \\ &= \left\lceil \frac{60\,000 \text{ изображений}}{128 \text{ изображений}} \right\rceil = \\ &= \lceil 468.75 \rceil = \\ &= 469. \end{aligned} \tag{8.4}$$

Перед началом обучения мы инициализируем параметры w и b всех нейронов случайными значениями[3]. Затем, перед первой эпохой обучения:

1. Перемешиваем и делим обучающий набор на пакеты по 128 изображений. Каждое из этих 128 изображений MNIST содержит 784 пиксела, составляющих входы x для передачи в нашу нейронную сеть. Именно этот шаг перемешивания соответствует слову *стохастический* (что означает *случайный*) в названии «стохастический градиентный спуск».
2. При прямом распространении информация о 128 изображениях обрабатывается каждым слоем сети и, наконец, выходной слой выводит результат — значения $\hat{y}$.

---

[1] Поскольку 60 000 не делится на 128 без остатка, этот 469-й пакет будет содержать только $0.75 \times 128 = 96$ изображений.

[2] Квадратные скобки, которые мы используем здесь и в уравнении 8.4, с отсутствующим горизонтальным хвостиком снизу, обозначают вычисления с округлением вверх до ближайшего целого. Например, число 468.75 округляется до 469.

[3] Подробнее об инициализации параметров случайными значениями рассказывается в главе 9.

3. Функция стоимости (например, на основе перекрестной энтропии) оценивает соответствие значений $\hat{y}$ истинным значениям y и вычисляет стоимость C для этого конкретного пакета из 128 изображений.
4. Чтобы минимизировать стоимость и тем самым улучшить оценки y сети для заданных x, выполняется градиентный спуск — часть стохастического градиентного спуска: каждый отдельный параметр w и b в сети корректируется пропорционально их вкладу в ошибку (то есть в стоимость) в этом пакете (обратите внимание, что корректировки масштабируются гиперпараметром скорости обучения η)[1].

Эти четыре шага составляют *цикл обучения*, как следует из рис. 8.5.

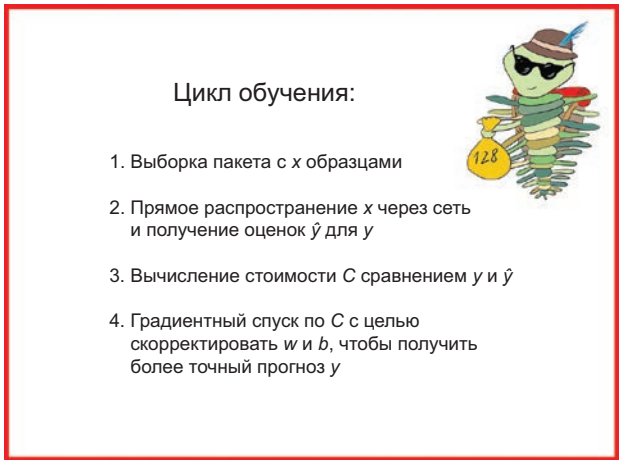


Рис. 8.5. Отдельный цикл обучения методом стохастического градиентного спуска. Размер пакета — это гиперпараметр, который можно изменять, но в данном конкретном случае пакет состоит из 128 цифр MNIST, как видно по изображению нашего трилобита-путешественника, несущего небольшой мешок данных

На рис. 8.6 показано, что циклы обучения повторяются до тех пор, пока не закончатся обучающие изображения в выборке. Выборка на шаге 1 выполняется *без замены*, то есть к концу эпохи алгоритм увидит каждое изображение только один раз, но между разными эпохами пакеты выбираются случайным образом. После 468 циклов последний пакет содержит только 96 образцов.

На этом завершается первая эпоха обучения. Если модель настроена на обучение на протяжении нескольких эпох, каждая следующая эпоха начинается с заполнения пула всеми 60 000 обучающими изображениями. Затем, как и в предыду-

[1] Величина коррекции, пропорциональная погрешности, вычисляется на этапе обратного распространения. Мы еще не знакомы с обратным распространением, но рассмотрим его в следующем разделе, так что держитесь крепче!

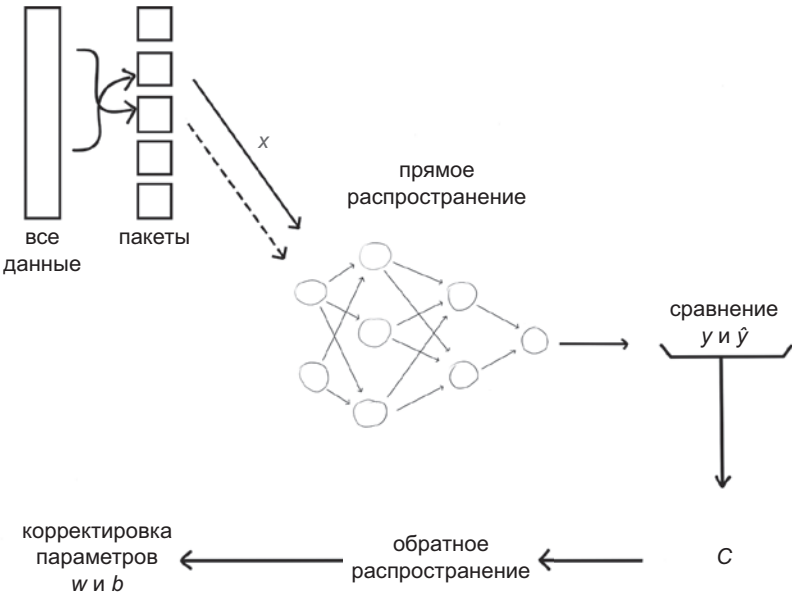


Рис. 8.6. Общая схема процесса обучения нейронной сети методом стохастического градиентного спуска. Весь набор данных перемешивается и разбивается на пакеты. Каждый пакет передается через сеть; результат $\hat{y}$ сравнивается с истинным y и вычисляется стоимость C; на этапе обратного распространения вычисляются градиенты и корректируются параметры w и b модели. Следующий пакет (обозначен пунктирной стрелкой) распространяется вперед и так далее, пока все партии не пройдут по сети. После передачи всех партий завершается одна эпоха, и процесс начинается снова с перемешивания набора обучающих данных

щую эпоху, выполняется еще 469 циклов стохастического градиентного спуска[1]. Обучение продолжается, пока не будет достигнуто желаемое количество эпох.

Кстати, общее *количество эпох* обучения сети — еще один гиперпараметр. Однако он является одним из самых простых для настройки:

- Если стоимость на проверочных данных снижается от эпохи к эпохе и последняя эпоха достигла наименьшей стоимости, можно попробовать увеличить число эпох.
- Как только стоимость на проверочных данных начинает расти, это явный признак *переобучения* модели на обучающих данных из-за того, что число эпох слишком высоко. (Подробнее о переобучении рассказывается в главе 9.)
- Существуют методы[2] автоматического мониторинга стоимости на обучающих и проверочных данных и прекращения обучения, когда что-то идет

[1] Поскольку выборка производится случайно, в каждой эпохе порядок следования обучающих изображений в 469 пакетах получается разным.

[2] См. *keras.io/callbacks/#earlystopping*.

не так. То есть можно заранее установить произвольно большое количество эпох и быть уверенными, что обучение будет продолжаться лишь до тех пор, пока стоимость на проверочных данных не прекратит улучшаться, и, конечно, до того, как модель начнет переобучаться!

КАК ИЗБЕЖАТЬ ЛОВУШКИ ЛОКАЛЬНОГО МИНИМУМА

Во всех примерах градиентного спуска, описанных выше в этой главе, наш трилобит-путешественник не сталкивался с препятствиями на пути к минимальной стоимости. Однако нет никаких гарантий, что этого никогда не произойдет. В действительности такие плавные склоны встречаются очень редко.

На рис. 8.7 трилобит-альпинист исследует стоимость в новой модели, которая используется для решения новой задачи. В этой задаче параметр p и стоимость C имеют более сложную связь. Чтобы нейронная сеть оценила y как можно точнее, градиентный спуск должен найти значения параметра, связанные с наименьшей достижимой стоимостью. Однако спускаясь вниз по склону из случайной начальной точки на верхнем кадре, трилобит попадает в *локальный минимум*. Как показано на среднем кадре, когда наш бесстрашный исследователь оказывается в точке локального минимума, шаг в любом направлении приводит к увеличению стоимости, поэтому слепой трилобит остается на месте, не замечая существования более глубокой долины — *глобального минимума*.

Но еще не все потеряно, друзья, потому что стохастический градиентный спуск снова приходит на помощь. Выборка пакетов может создавать эффект сглаживания кривой стоимости, как показано пунктирной линией на нижнем кадре на рис. 8.7. Это сглаживание обусловлено искажениями, возникающими при оценке градиента на пакетах небольшого размера (по сравнению со всем набором данных). Фактический градиент (угол наклона) в локальном минимуме действительно равен нулю, однако оценки градиента по небольшим подмножествам данных не позволяют получить полную картину и могут дать неточный результат, заставляя трилобита сделать шаг влево, так как он думает, что градиент (уклон) существует, хотя в действительности его нет. Эти искажения и неточности — парадоксально хорошая штука! Неправильная оценка градиента может привести к тому, что шаг окажется достаточно большим, чтобы трилобит перешагнул через препятствие, покинул локальную долину и продолжил свой путь вниз по склону. То есть при многократной оценке градиента на пакетах происходит сглаживание искажений и появляется возможность избежать локальных минимумов. Таким образом, хотя каждый пакет сам по себе не имеет полной информации о кривой стоимости, которую можно наблюдать на большом количестве пакетов, это обстоятельство работает на нас.

Так же как в случае с гиперпараметром скорости обучения η, размер пакетов тоже имеет оптимальное значение. При использовании пакетов большого размера оценка градиента функции стоимости будет намного точнее. Благодаря

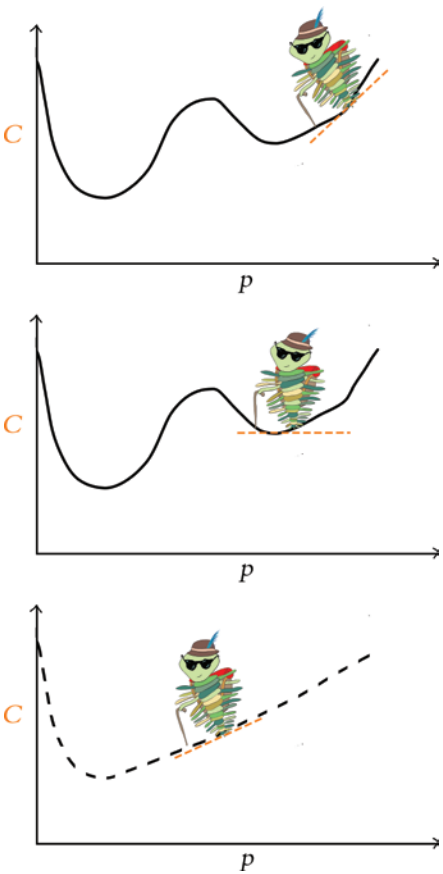


Рис. 8.7. Трилобит, использующий обычный метод градиентного спуска, начав движение со случайной начальной точки (верхний кадр), натыкается на локальный минимум стоимости (средний кадр). Вернувшись к стохастическому градиентному спуску (нижний кадр), дерзкий трилобит способен выйти из локального минимума и достичь глобального минимума

этому трилобит получит более точное представление о градиенте в непосредственной близости и сможет сделать шаг (пропорциональный) в направлении максимально крутого спуска. При этом увеличивается риск оказаться в ловушке локального минимума, как было описано выше[1]. Кроме того, модель может не уместиться в памяти компьютера, и время вычисления каждой итерации градиентного спуска окажется очень большим.

Напротив, при использовании пакетов слишком маленького размера оценки градиентов могут сильно искажаться (потому что для оценки градиента всего набора данных используется очень небольшое подмножество) и соответствующий путь вниз по склону будет излишне извилист; обучение займет больше времени из-за этих беспорядочных шагов. Кроме того, недоиспользованными

[1] Стоит отметить, что здесь важную роль играет скорость обучения. Если протяженность локального минимума окажется *меньше* размера шага, трилобит сможет перешагнуть локальный минимум, подобно тому как мы перешагиваем трещины на асфальте.

останутся память и вычислительные ресурсы компьютера[1]. С учетом этого мы предлагаем следующие практические рекомендации по поиску оптимального размера пакетов:

- для начала выберите размер пакета 32;
- если пакет слишком велик и не умещается в памяти компьютера, попробуйте уменьшить его размер на одну степень двойки (например, с 32 до 16);
- если модель демонстрирует хорошую способность к обучению (то есть стоимость постоянно снижается), но на каждую эпоху уходит очень много времени и известно, что в компьютере имеется достаточно ОЗУ[2], попробуйте увеличить размер пакета. Чтобы избежать попадания в локальные минимумы, мы не советуем использовать размеры пакетов больше 128.

ОБРАТНОЕ РАСПРОСТРАНЕНИЕ

В большинстве моделей машинного обучения стохастический градиентный спуск прекрасно справляется с корректировкой параметров и минимизацией затрат, но в моделях глубокого обучения, в частности, существует дополнительное препятствие: необходимо иметь возможность эффективно регулировать параметры *в нескольких слоях* искусственных нейронов. Для этого стохастический градиентный спуск объединяют с методикой, которая называется *обратным распространением*.

Обратное распространение — это элегантное применение «цепного правила» из дифференциального и интегрального исчисления[3]. Как показано внизу на рис. 8.6 и как следует из самого названия, обратное распространение происходит через нейронную сеть в направлении, обратном прямому распространению. Если при прямом распространении через последовательность слоев переносится информация о входном значении x, чтобы получить оценку $\hat{y}$, близкую к y, то при обратном распространении через последовательность слоев (*в обратном направлении*) переносится информация о стоимости C,

[1] Стохастический градиентный спуск с размером партии 1 называют *онлайн-обучением*. Стоит отметить, что это не самый быстрый метод с вычислительной точки зрения. Умножение матриц, выполняемое в каждом цикле обучения на пакетах, высоко оптимизировано, поэтому обучение может протекать на несколько порядков быстрее при использовании пакетов среднего размера по сравнению с *онлайн-обучением*.

[2] В Unix и Unix-подобных операционных системах, включая macOS, использование оперативной памяти можно оценить, выполнив команду `top` или `htop` в окне терминала.

[3] Чтобы описать математический аппарат, лежащий в основе обратного распространения, необходимо углубиться в тему вычисления частной производной. Мы приветствуем желание глубже познакомиться с красотой обратного распространения, но понимаем, что дифференциальное и интегральное исчисление может быть не самой интересной темой. Поэтому мы поместили описание математики обратного распространения в приложение Б.

и главной целью является снижение стоимости путем корректировки параметров нейронов во всей сети.

Подробное описание механики обратного распространения мы вынесли в Приложение Б. Однако важно понимать (в общих чертах), что делает алгоритм обратного распространения: любая модель нейронной сети инициализируется случайными значениями параметров w и b (порядок инициализации подробно описан в главе 9). То есть в начале обучения, когда вводится первое значение x, сеть выводит случайный прогноз $\hat{y}$. Едва ли этот прогноз будет хоть сколько-нибудь точным, и стоимость, связанная с этим случайным прогнозом, наверняка получится очень высокой. После этого сеть должна скорректировать веса, чтобы минимизировать стоимость, — в этом заключается суть машинного обучения. Для этого используется обратное распространение, в ходе которого вычисляется *градиент* функции стоимости по отношению к каждому весу в сети.

Следуя аналогии с альпинизмом, функцию стоимости можно сравнить с пешеходной тропой, по которой наш трилобит пытается добраться до базового лагеря. На каждом шаге он находит градиент (наклон) функции стоимости и перемещается *вниз* по этому градиенту. Это движение соответствует изменению веса: регулируя вес пропорционально градиенту функции стоимости *с учетом этого веса*, обратное распространение корректирует вес в направлении, ведущем к снижению стоимости.

Вспомнив «самое важное уравнение», показанное на рис. 6.7 ($w \cdot x + b$), и тот факт, что нейронные сети накапливают информацию, распространяющуюся вперед через слои, нетрудно догадаться, что любой вес в сети вносит свой вклад в конечный выходной сигнал и, следовательно, стоимость C. Используя обратное распространение, мы перемещаемся слой за слоем назад по сети, начиная со стоимости в выходном слое, и находим градиент каждого отдельного параметра, который затем можно использовать для корректировки параметра вверх или вниз (с шагом, соответствующим скорости обучения) — в зависимости от того, какое из двух направлений приведет к уменьшению стоимости.

Мы признаем, что это не самый простой раздел в книге. Если вы сможете вынести из него что-то одно, пусть это будет следующее: обратное распространение вычисляет относительный вклад каждого отдельного параметра в общую стоимость, а затем соответствующим образом корректирует его. То есть сеть итеративно снижает стоимость и... учится!

НАСТРОЙКА ЧИСЛА СКРЫТЫХ СЛОЕВ И НЕЙРОНОВ

По аналогии со скоростью обучения и размера пакета, количество скрытых слоев в нейронной сети является гиперпараметром. И, как и в случае с двумя предыдущими гиперпараметрами, число слоев в сети тоже имеет оптимальное значение. В этой книге мы не раз говорили, что с каждым дополнительным скрытым слоем

сеть глубокого обучения получает возможность выявлять все более абстрактные представления. В этом заключается главное преимущество добавления слоев.

Недостаток увеличения числа слоев заключается в уменьшении эффективности обратного распространения: как показывает график скорости обучения слоев в сети с пятью скрытыми слоями на рис. 8.8, наибольшее влияние обратное распространение оказывает на параметры скрытого слоя, ближайшего к выходу $\hat{y}$[1]. Чем дальше слой от $\hat{y}$, тем слабее влияют его параметры на общую стоимость. То есть пятый слой, ближайший к выходу $\hat{y}$, обучается быстрее, потому что его веса имеют наибольшие градиенты. Напротив, третий скрытый слой, находящийся в нескольких слоях от выходного слоя, обучается на порядок медленнее пятого.

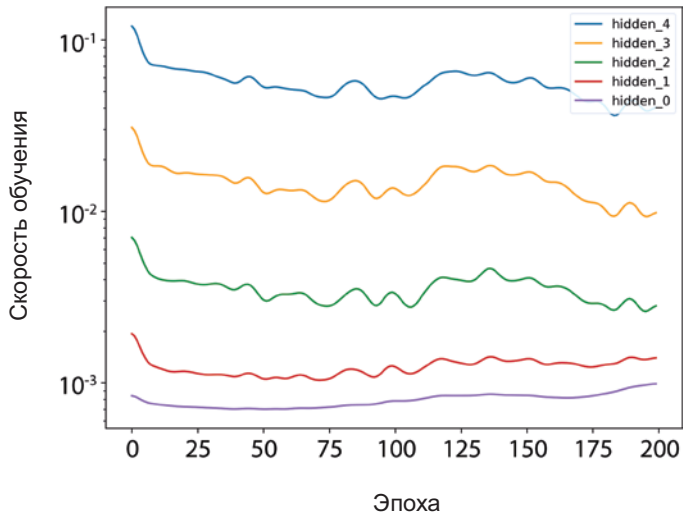


Рис. 8.8. Скорость обучения глубокой сети с пятью скрытыми слоями в разные эпохи. Пятый скрытый слой, который ближе всего к выходу $\hat{y}$, обучается на порядок быстрее, чем третий

С учетом всего вышесказанного, вот наши рекомендации по выбору числа скрытых слоев в сети:

- Чем более абстрактно истинное значение y, которое сеть должна аппроксимировать, тем больше пользы можно извлечь из наличия дополнительных скрытых слоев, поэтому мы советуем начинать с двух-четырех скрытых слоев.
- Если уменьшение количества слоев не влечет за собой увеличение стоимости, достигаемой на проверочном наборе данных, то уменьшите это число.

[1] Если вам интересно, как мы получили график на рис. 8.8, загляните в блокнот Jupyter *measuring_speed_of_learning.ipynb*.

Следуя принципу *бритвы Оккама*, более простая архитектура сети, способная обеспечить желаемый результат, является наилучшей; она будет обучаться быстрее и потребует меньше вычислительных ресурсов.

- С другой стороны, если увеличение количества слоев снижает стоимость на проверочных данных, тогда вам определенно стоит увеличить его!

Кроме глубины сети есть еще один гиперпараметр модели — число нейронов в каждом слое. Если в сети много слоев, значит, в них можно настроить количество нейронов. Сначала эта перспектива может показаться пугающей, но не волнуйтесь: если число нейронов окажется больше необходимого, это лишь немного увеличит вычислительную сложность сети; если число нейронов окажется немного меньше, это почти не повлияет на ее точность.

По мере накопления опыта в создании и обучении моделей глубокого обучения для решения разных задач вы начнете лучше понимать, сколько нейронов использовать в том или ином слое. В зависимости от конкретных моделируемых данных в них может иметься много низкоуровневых признаков, которые желательно выявить, и в этом случае лучше иметь больше нейронов в первых слоях сети. Если, напротив, в данных имеется много высокоуровневых признаков, тогда может оказаться предпочтительнее увеличить число нейронов в более поздних слоях. Мы сами обычно экспериментируем с количеством нейронов в каждом слое, изменяя его на степени двойки. Если удвоение числа нейронов с 64 до 128 заметно улучшает точность модели, удвойте его. Но имейте в виду, что, перефразируя принцип бритвы Оккама, если уменьшение числа нейронов с 64 до 32 не ухудшает точности модели, то это, вероятно, верное решение, потому что оно уменьшает вычислительную сложность модели без видимых отрицательных эффектов.

СЕТЬ ПРОМЕЖУТОЧНОЙ ГЛУБИНЫ НА ОСНОВЕ KERAS

В завершение этой главы воплотим новые теоретические познания в нейронную сеть и посмотрим, сможем ли мы превзойти предыдущую модель *shallow_net_in_keras.ipynb* в классификации рукописных цифр.

Первые несколько этапов создания сети промежуточной глубины в блокноте Jupyter *intermediate_net_in_keras.ipynb* идентичны этапам создания ее предшественницы — неглубокой сети. Сначала загружаются те же самые зависимости Keras, точно так же вносится и обрабатывается набор данных MNIST. Как можно увидеть в листинге 8.1, самое интересное начинается там, где определяется архитектура нейронной сети.

Листинг 8.1. Код, определяющий архитектуру нейронной сети с промежуточной глубиной

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Первая строка в этом фрагменте кода, `model = Sequential()`, та же, что и в предыдущей сети (листинг 5.6); это экземпляр объекта модели нейронной сети. В следующей строке начинаются расхождения. В ней мы заменили функцию активации `sigmoid` в первом скрытом слое функцией `relu`, как было рекомендовано в главе 6. Все остальные параметры первого слоя, кроме функции активации, остались прежними: он все так же состоит из 64 нейронов, и прежней осталась размерность входного слоя — 784 нейрона.

Другое существенное изменение в листинге 8.1 по сравнению с неглубокой архитектурой в листинге 5.6 заключается в наличии второго скрытого слоя искусственных нейронов. Вызовом метода `model.add()` мы без всяких усилий добавляем второй слой `Dense` с `64` нейронами `relu`, оправдывая слово *intermediate* (промежуточная) в имени блокнота. Вызвав `model.summary()`, можно увидеть, как показано на рис. 8.9, что этот дополнительный слой добавляет 4160 дополнительных обучаемых параметров, по сравнению с неглубокой архитектурой (см. рис. 7.5). Параметры можно разбить на:

- 4096 весов, соответствующих связям каждого из 64 нейронов во втором скрытом слое с каждым из 64 нейронов в первом скрытом слое ($64 \times 64 = 4096$);
- плюс 64 смещения, по одному для каждого нейрона во втором скрытом слое;
- в результате получается 4160 параметров: $n_{\text{параметров}} = n_w + n_b = 4096 + 64 = 4160$.

Помимо изменений в архитектуре модели мы также изменили параметры компиляции модели, как показано в листинге 8.2.

Листинг 8.2. Код компиляции нейронной сети с промежуточной глубиной

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

Рис. 8.9. Сводная информация о модели из блокнота Jupyter intermediate_net_in_keras.ipynb

Эти строки из листинга 8.2:

- задают функцию стоимости на основе перекрестной энтропии: `loss='categorical_crossentropy'` (в неглубокой сети использовалась квадратичная стоимость `loss='mean_squared_error'`);
- задают метод стохастического градиентного спуска для минимизации стоимости: `optimizer=SGD`;
- определяют гиперпараметр скорости обучения: `lr=0.1`[1];
- указывают, что в дополнение к обратной связи о потерях, которая предлагается библиотекой Keras по умолчанию, мы также хотим получить обратную связь о точности модели: `metrics=['accuracy']`[2].

Наконец, мы обучаем промежуточную сеть, выполняя код в листинге 8.3.

Листинг 8.3. Код обучения нейронной сети с промежуточной глубиной

```
model.fit(X_train, y_train,
          batch_size=128, epochs=20,
          verbose=1,
          validation_data=(X_valid, y_valid))
```

Единственное, что изменилось в обучении промежуточной сети по сравнению с неглубокой сетью (см. листинг 5.7), — это уменьшение на порядок гиперпараметра `epochs` с `200` до `20`. Как вы увидите далее, более эффективная промежуточная архитектура требует намного меньше эпох для обучения.

```
Epoch 1/20
60000/60000 [==============================] - 1s 15us/step - loss: 0.4744 - acc: 0.8637 - val_loss: 0.2686 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.2414 - acc: 0.9289 - val_loss: 0.2004 - val_acc: 0.9404
Epoch 3/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1871 - acc: 0.9452 - val_loss: 0.1578 - val_acc: 0.9521
Epoch 4/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1538 - acc: 0.9551 - val_loss: 0.1435 - val_acc: 0.9574
```

Рис. 8.10. Качество прогнозирования нейронной сети промежуточной глубины в первые четыре эпохи обучения

[1] Вы можете попробовать увеличить скорость обучения на несколько порядков, а затем уменьшить ее на несколько порядков и понаблюдать, как это повлияет на обучение.

[2] Потеря является наиболее важным показателем, позволяющим увидеть, как изменяется качество модели с течением эпох, но конкретное значение потери зависит от характеристик данной модели и, как правило, не поддается интерпретации и не может сравниваться у разных моделей. Поэтому даже зная, что потери должны быть как можно ближе к нулю, часто очень сложно понять, насколько близка к нулю потеря для конкретной модели. Точность, напротив, легко интерпретируется и легко обобщается: мы точно знаем, что это значит (например, «неглубокая нейронная сеть правильно классифицировала 86 процентов рукописных цифр в проверочном наборе данных»), и можем сравнить с точностью любой другой модели («точность 86 процентов хуже, чем точность нашей глубокой нейронной сети»).

На рис. 8.10 представлены результаты первых четырех эпох обучения сети. Как вы наверняка помните, наша неглубокая архитектура достигла плато на уровне 86%-й точности на проверочных данных после 200 эпох. Сеть промежуточной глубины значительно превзошла ее: как показывает поле `val_acc`, сеть достигла 92.34%-й точности *уже после первой эпохи обучения*. После третьей эпохи точность превысила 95%, а к 20-й эпохе, похоже, достигла плато на уровне около 97.6%. Мы серьезно продвинулись вперед!

Разберем подробнее вывод `model.fit()`, показанный на рис. 8.10:

- Индикатор процесса, показанный ниже, заполняется в течение 469 «циклов обучения» (см. рис. 8.5):

```
60000/60000 [==============================]
```

- `1s 15us/step` означает, что на все 469 циклов обучения в первую эпоху потребовалась 1 секунда, в среднем по 15 микросекунд на цикл.
- `loss` показывает среднюю стоимость на обучающих данных для эпохи. Для первой эпохи она равна `0.4744` и от эпохи к эпохе уверенно уменьшается методами стохастического градиентного спуска (SGD) и обратного распространения, а в конечном итоге уменьшается до `0.0332` к двадцатой эпохе.
- `acc` — точность классификации на обучающих данных в данную эпоху. Модель правильно классифицировала 86.37% после первой эпохи и достигла уровня выше 99% к двадцатой. Поскольку модель может переобучиться, не следует особо удивляться высокой точности в этом параметре.
- К счастью, стоимость на проверочном наборе данных (`val_loss`), как правило, тоже уменьшается и в конечном итоге достигает плато на уровне `0.08` в ходе последних пяти эпох обучения.
- Одновременно с уменьшением стоимости на проверочных данных растет точность (`val_acc`). Как уже упоминалось, точность на проверочных данных составила 97.6%, что значительно выше 86% у нашей неглубокой сети.

ИТОГИ

В этой главе мы проделали большую работу. Сначала мы узнали, как нейронная сеть с фиксированными параметрами обрабатывает информацию. Затем разобрались с взаимодействующими методами — функциями стоимости, стохастическим градиентным спуском и обратным распространением, которые позволяют корректировать параметры сети для аппроксимации любого истинного значения y, имеющего непрерывное отношение с некоторым входом x. Попутно мы познакомились с несколькими гиперпараметрами, включая скорость обучения, размер пакета и количество эпох обучения, а также с практическими правилами настройки каждого из них. В завершение главы мы применили новые знания

для создания нейронной сети промежуточной глубины, которая значительно превзошла предыдущую неглубокую сеть на той же задаче классификации рукописных цифр. Далее мы познакомимся с методами улучшения стабильности искусственных нейронных сетей по мере их углубления, что позволит нам разработать и обучить полновесную модель глубокого обучения.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым.

- параметры:
  - вес w;
  - смещение b;
- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
- входной слой;
- скрытый слой;
- выходной слой;
- типы слоев:
  - плотный (полносвязанный);
  - softmax;
- функции стоимости (потерь):
  - квадратичная (среднеквадратичная ошибка);
  - перекрестная энтропия;
- прямое распространение;
- обратное распространение;
- оптимизаторы:
  - стохастический градиентный спуск;
- гиперпараметры оптимизаторов:
  - скорость обучения η;
  - размер пакета.

9

СОВЕРШЕНСТВОВАНИЕ ГЛУБОКИХ СЕТЕЙ

В главе 6 мы детально исследовали отдельные искусственные нейроны. В главе 7 объединили их в сеть, организовав прямое распространение некоторых входных данных x через сеть для получения некоторого прогноза $\hat{y}$. А совсем недавно, в главе 8, мы узнали, как количественно оценить ошибки сети (сравнить оценку $\hat{y}$ с истинным значением y с помощью функции стоимости), а также как минимизировать эти ошибки (настроить параметры сети w и b с применением оптимизаторов — стохастического градиентного спуска и обратного распространения).

Теперь мы рассмотрим типичные препятствия, встречающиеся на пути создания высококачественных нейронных сетей, и методы их преодоления. Мы применим эти идеи при разработке нашей первой глубокой нейронной сети[1]. Объединив увеличенную глубину сети с новыми приемами, мы посмотрим, можно ли превзойти более простые и мелкие архитектуры из предыдущих глав в точности классификации рукописных цифр.

ИНИЦИАЛИЗАЦИЯ ВЕСОВ

В главе 8 мы познакомились с эффектом насыщения нейронов (см. рис. 8.1), когда очень низкие или очень высокие значения z ухудшают способность нейрона к обучению. Там же было предложено решение в виде функции стоимости на основе перекрестной энтропии. Перекрестная энтропия существенно ослабляет влияние эффекта насыщения нейронов, но, объединив ее с вдумчивой *инициализацией весов*, можно еще больше уменьшить вероятность появления эффекта насыщения. Как отмечалось в сноске в главе 1, применение усовершенствованных методов

[1] Как рассказывалось в главе 4, *глубокой* называется нейронная сеть, состоящая по крайней мере из трех скрытых слоев.

инициализации весов обеспечило существенный скачок в развитии сферы глубокого обучения: это одно из знаковых теоретических достижений, сделанных между LeNet-5 (см. рис. 1.11) и AlexNet (см. рис. 1.17), которые значительно расширили диапазон задач, решаемых с помощью искусственных нейронных сетей. В этом разделе мы поэкспериментируем с несколькими подходами к инициализации весов, чтобы вы смогли понять, насколько они эффективны.

Описывая порядок обучения нейронной сети в главе 8, мы упоминали, что параметры w и b инициализируются случайными значениями, из-за чего начальная аппроксимация сети оказывается далекой от цели и, соответственно, имеет высокую начальную стоимость C. Вообще говоря, нет большой нужды подробно рассматривать эту проблему, потому что за кулисами библиотека Keras создает модели TensorFlow, которые инициализируются вполне разумными значениями w и b. Тем не менее мы обсудим ее, чтобы вы не только знали о существовании еще одного метода предотвращения насыщения нейронов, но и восполнили пробел в вашем понимании особенностей обучения нейронных сетей. Библиотека Keras берет на себя всю тяжелую работу по выбору начальных значений, и это главное ее преимущество; но вы должны понимать, что возможно, а иногда необходимо изменить эти значения по умолчанию, чтобы добиться лучшего соответствия вашей задаче.

Чтобы чтение этого раздела проходило более увлекательно, мы советуем использовать наш блокнот Jupyter *weight_initialization.ipynb*. Как показано в следующем фрагменте кода, блокнот зависит от NumPy (библиотека числовых операций), matplotlib (библиотека для построения графиков) и нескольких методов Keras, которые мы подробно рассмотрим в этом разделе.

```
import numpy as np
import matplotlib.pyplot as plt
from keras import Sequential
from keras.layers import Dense, Activation
from keras.initializers import Zeros, RandomNormal
from keras.initializers import glorot_normal, glorot_uniform
```

В этом блокноте мы моделируем 784-пиксельные значения, которые служат входными данными для одного полносвязанного слоя искусственных нейронов. Такая размерность входных данных, конечно же, была выбрана по образу и подобию наших любимых цифр из коллекции MNIST (рис. 5.3). Для полносвязанного слоя мы выбрали достаточно большое число нейронов (256), чтобы потом, когда мы будем строить графики, у нас имелось достаточно данных:

```
n_input = 784
n_dense = 256
```

Основной темой этого раздела является инициализация параметров сети w и b. Прежде чем начать передавать обучающие данные в сеть, желательно определить разумные начальные значения параметров. Тому есть две причины.

1. Большие значения w и b, как правило, приводят к большим значениям z и, соответственно, к появлению эффекта насыщения нейронов (см. график на рис. 8.1).
2. Большие значения параметров означают, что сеть имеет четкое представление о том, как входные данные x связаны с истинными значениями y, но было бы странно строить любые предположения до обучения на фактических данных.

Нулевые значения параметров, с другой стороны, предполагают весьма расплывчатое представление о взаимосвязях между x и y. Проводя аналогию со сказкой «Три медведя», наша цель — уподобиться Машеньке и начать обучение со сбалансированными и *пригодными к обучению* исходными значениями параметров. По этой причине, проектируя архитектуру нейронной сети, мы использовали метод `Zeros()` для инициализации нейронов полносвязанного слоя $b = 0$:

```
b_init = Zeros()
```

Продолжая рассуждения, начатые в предыдущем абзаце, можно подумать, что веса w сети тоже следует инициализировать нулями. Но в действительности это привело бы к катастрофе: при одинаковых значениях весов и смещений многие нейроны в сети будут одинаково интерпретировать одни и те же входные данные x, мешая стохастическому градиентному спуску (SGD) определять индивидуальные настройки параметров, способствующие снижению стоимости C. Продуктивнее было бы инициализировать веса разными значениями из некоторого диапазона, чтобы каждый нейрон обрабатывал входные данные x уникально и давал алгоритму SGD широкий выбор начальных точек для аппроксимации y. По теории вероятностей, начальные результаты некоторых нейронов могут внести верный вклад в разумное отображение x в y. Сначала этот вклад будет слабым, но у SGD появится шанс поэкспериментировать с ним и определить, может ли он способствовать снижению стоимости C между прогнозируемым $\hat{y}$ и целевым значением y.

Как отмечалось ранее (например, при обсуждении рис. 7.5 и 8.9), подавляющее большинство параметров в типичной сети составляют веса, а на смещения приходится относительно небольшая их часть. Поэтому допустимо (в действительности это наиболее распространенная практика) инициализировать смещения нулями, а веса — случайными значениями, *близкими* к нулю. Один из самых простых способов сгенерировать случайные значения, близкие к нулю, — выбирать их из стандартного нормального распределения[1], как в листинге 9.1.

Листинг 9.1. Инициализация весов значениями, выбираемыми из стандартного нормального распределения

```
w_init = RandomNormal(stddev=1.0)
```

[1] Нормальное распределение также называют распределением Гаусса или, в разговорной речи, «колоколообразной кривой» из-за соответствующей формы графика. *Стандартное нормальное* распределение — это нормальное распределение со средним значением 0 и стандартным отклонением 1.

Чтобы оценить влияние выбранного способа инициализации весов, в листинге 9.2 создается архитектура нейронной сети с единственным полносвязанным слоем сигмоидных нейронов.

Листинг 9.2. Архитектура с единственным полносвязанным слоем сигмоидных нейронов

```
model = Sequential()
model.add(Dense(n_dense,
                input_dim=n_input,
                kernel_initializer=w_init,
                bias_initializer=b_init))
model.add(Activation('sigmoid'))
```

Как и во всех предыдущих примерах, модель создается с помощью `Sequential()`. Затем используется метод `add()` для создания одного полносвязанного слоя со следующими параметрами:

- 256 нейронов (`n_dense`);
- 784 входа (`n_input`);
- в `kernel_initializer` передается массив `w_init` для инициализации весов сети желаемыми значениями — в данном случае выбранными из стандартного нормального распределения;
- в `bias_initializer` передается массив `b_init` для инициализации смещений нулевыми значениями.

Чтобы упростить корректировку параметров далее в этом разделе, отдельно добавляем в слой сигмоидную функцию активации вызовом `Activation('sigmoid')`.

После настройки сети вызывается метод `random()` из библиотеки NumPy, чтобы сгенерировать 784 «значения пикселов» — случайные числа с плавающей точкой из диапазона [0.0, 1.0):

```
x = np.random.random((1,n_input))
```

Далее вызывается метод `predict()` для прямого распространения x через один слой и получения активаций `a`:

```
a = model.predict(x)
```

В заключение генерируется гистограмма, иллюстрирующая распределение активаций[1]:

```
_ = plt.hist(np.transpose(a))
```

[1] Для тех, кому интересно, что означает символ подчеркивания (`_ =`), поясняем, что этот прием помогает сохранить блокнот Jupyter более аккуратным и просто вывести график, без создания объекта, хранящего этот график.

Полученные нами результаты показаны на рис. 9.1. Ваши результаты будут немного отличаться от наших из-за метода `random()`, который использовался для получения входных значений, но в целом они должны выглядеть примерно похожими.

Как и ожидалось (см. рис. 6.9), активации a на выходе слоя сигмоидных нейронов ограничены диапазоном от 0 до 1. Однако в их распределении отмечается нежелательная закономерность — большая их часть сосредоточена по краям диапазона: они примыкают либо к 0, либо к 1. Это указывает на то, что при использовании нормального распределения для инициализации весов w слоя наши искусственные нейроны склонны производить большие значения z. Это нежелательно по двум причинам, упомянутым выше в этом разделе:

1. Подавляющее большинство нейронов в слое подвержено эффекту насыщения.
2. Нейроны выражают твердое мнение о том, как x влияет на y *еще до* обучения на данных.

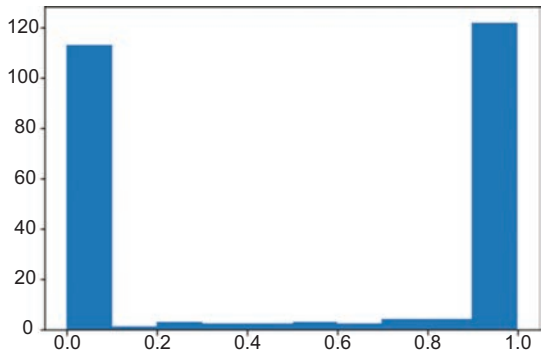


Рис. 9.1. Гистограмма распределения активаций на выходе слоя сигмоидных нейронов с весами, инициализированными с использованием стандартного нормального распределения

К счастью, эту проблему легко устранить, инициализировав веса сети значениями, выбранными из альтернативных распределений.

РАСПРЕДЕЛЕНИЯ КСАВЬЕ ГЛОРО

В сфере глубокого обучения большой популярностью для выборки начальных значений весов пользуются распределения, разработанные Ксавье Глоро (Xavier Glorot) и Йошуа Бенжио[1], портрет которого представлен на рис. 1.10.

[1] *Glorot X. & Bengio Y.* (2010). «Understanding the difficulty of training deep feedforward neural networks». Proceedings of Machine Learning Research, 9, 249–56.

Эти *распределения Глоро*, как их обычно называют[1], адаптированы так, что выборка из них приводит к тому, что нейроны первоначально выводят небольшие значения *z*. Давайте рассмотрим их поближе. Заменим код, производящий выборку из стандартного нормального распределения (листинг 9.1) в блокноте *weight_initialization.ipynb* строкой из листинга 9.3. В данном случае мы использовали *нормальное распределение Глоро*[2].

Листинг 9.3. Инициализация весов значениями, выбираемыми из нормального распределения Глоро

```
w_init = glorot_normal()
```

Перезапустив блокнот[3], вы должны увидеть распределение активаций, аналогичное гистограмме на рис. 9.2.

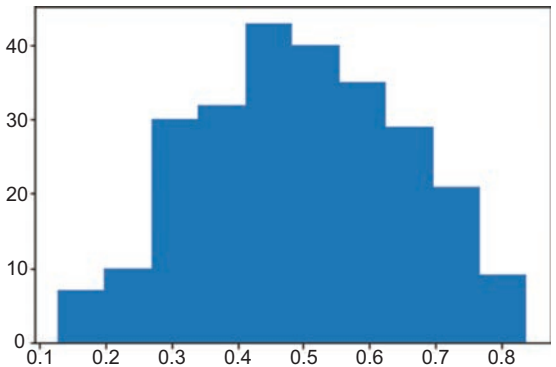


Рис. 9.2. Гистограмма распределения активаций на выходе слоя сигмоидных нейронов с весами, инициализированными с использованием нормального распределения Глоро

На этот раз, в отличие от рис. 9.1, активации a, вычисленные слоем сигмоидных нейронов, сосредоточились вблизи среднего значения ~0.5, и лишь несколько из них оказались на краях сигмоиды (то есть со значением меньше 0.1 или больше 0.9). Это хорошая отправная точка для нейронной сети, потому что теперь:

1. Лишь несколько нейронов (если таковые вообще имеются) оказались насыщенными.

---

[1] Некоторые называют их *распределениями Ксавье*.

[2] Нормальное распределение Глоро — это усеченное нормальное распределение с центром в точке 0 и стандартным отклонением $\sqrt{\frac{2}{n_{in}+n_{out}}}$, где n_{in} — количество нейронов в предыдущем слое, а n_{out} — количество нейронов в последующем слое.

[3] Выберите пункт Kernel (Ядро) в меню блокнота Jupyter и затем пункт Restart & Run All (Перезапустить все). В этом случае все вычисления будут выполнены с самого начала и не будут повторно использованы параметры, настроенные при предыдущем запуске.

2. Подавляющее большинство нейронов имеет весьма расплывчатое мнение о том, как x влияет на y, что — до начала обучения на данных — вполне разумно.

Как будет показано в этом разделе, один из смущающих аспектов инициализации весов состоит в том, что, если мы хотим получить нормально распределенные (и мы этого добились!) значения a, возвращаемые слоем искусственных нейронов, не следует выбирать начальные веса из стандартного нормального распределения.

Помимо нормального распределения Глоро существует также *равномерное распределение Глоро*[1]. Выбор того или иного распределения Глоро для инициализации весов модели, как правило, не оказывает существенного влияния. Вы можете перезапустить блокнот и попробовать выбирать значения из равномерного распределения Глоро, установив `w_init` равным `glorot_uniform()`. В этом случае вы должны получить гистограмму активаций, более или менее неотличимую от гистограммы на рис. 9.2.

Заменив в блокноте *weight_initialization.ipynb* функцию активации sigmoid (листинг 9.2) функцией tanh (`Activation('tanh')`) или ReLU (`Activation('relu')`), можно увидеть, к каким последствиям в этих случаях приводит инициализация весов значениями, взятыми из стандартного нормального распределения и распределения Глоро. Как показано на рис. 9.3, независимо от выбранной функции активации, инициализация весов значениями из стандартного нормального распределения приводит к получению активаций a, сосредоточенных по краям, в отличие от инициализации значениями, полученными из распределения Глоро.

Для большей уверенности желающие могут ближе познакомиться с подходами к инициализации параметров в документации библиотеки Keras с описанием слоев разных типов, но, как мы уже сказали, по умолчанию она инициализирует смещения значениями 0, а веса — выбирая значения из распределения Глоро.

Инициализация распределением Глоро является, пожалуй, наиболее популярным методом, но есть и другие неплохие варианты, такие как инициализация Хе (He)[2] и инициализация Лекуна[3]. По своему опыту можем сказать, что различия между этими методами минимальна или даже незаметна.

---

[1] Равномерное распределение Глоро находится в диапазоне $[-l, l]$, где $l = \sqrt{\frac{6}{n_{in} + n_{out}}}$.

[2] *He Y.* et al. (2015). «Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification». *arXiv: 1502.01852*.

[3] *LeCun Y.* et al. (1998). «Efficient backprop. In G. Montavon et al. (Eds.)». Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, 7700 (pp. 355–65). Berlin: Springer.

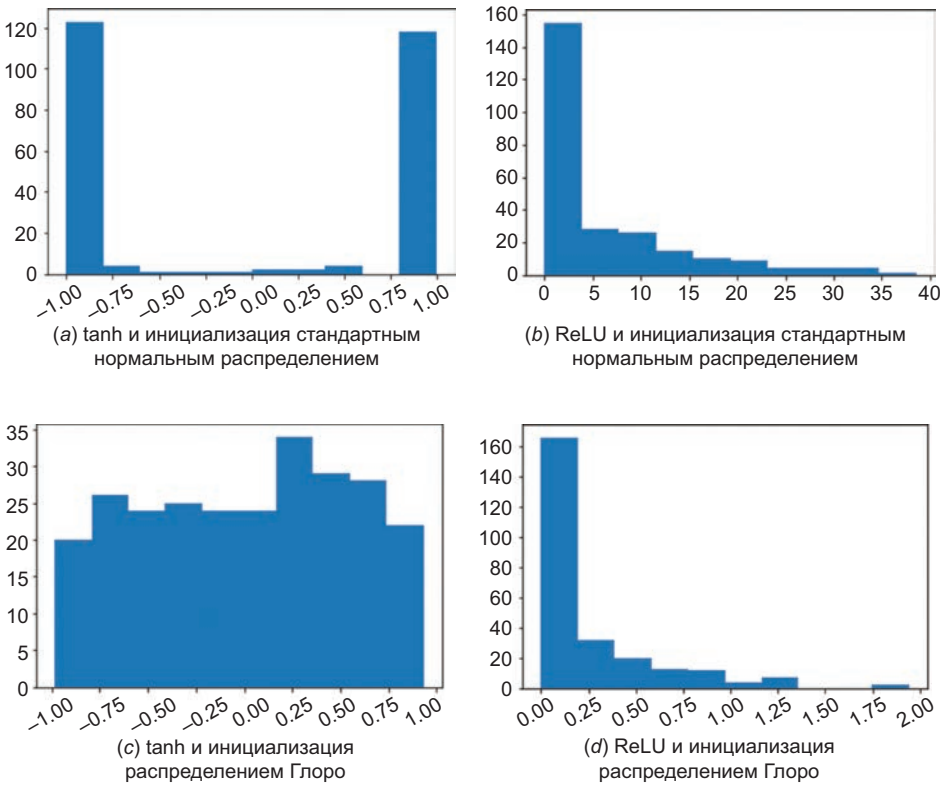


Рис. 9.3. Активации, получаемые полносвязанным слоем с 256 нейронами при использовании разных функций активации (tanh и ReLU) и способов инициализации весов (стандартным нормальным и равномерным распределениями Глоро). Обратите внимание, что даже при кажущейся схожести гистограмм (b) и (d), инициализация стандартным нормальным распределением привела к большим значениям активации (достигающим 40), а инициализация распределением Глоро привела к активациям ниже 2

НЕСТАБИЛЬНОСТЬ ГРАДИЕНТОВ

Еще одна проблема, возникающая в искусственных нейронных сетях и обостряющаяся с увеличением числа скрытых слоев, — *нестабильность градиентов*. Нестабильность градиентов может иметь исчезающий или взрывной характер. Мы рассмотрим оба варианта по очереди, а затем обсудим решение, называемое пакетной нормализацией.

ИСЧЕЗАЮЩИЕ ГРАДИЕНТЫ

Напомним, что при использовании стоимости C между предсказанными сетью значениями $\hat{y}$ и истинными y, как показано на рис. 8.6, обратное распростра-

нение протекает от выходного слоя к входному и регулирует параметры сети, стремясь минимизировать стоимость. Как было показано на примере трилобита-альпиниста на рис. 8.2, каждый параметр корректируется пропорционально его *градиенту* стоимости: если, например, градиент параметра большой и положительный, это означает, что данный параметр вносит *большой* вклад в стоимость, и, следовательно, пропорциональное его *уменьшение* будет соответствовать снижению стоимости[1].

Параметры скрытого слоя, находящегося ближе к выходному слою, имеют более прямую связь со стоимостью. Чем дальше скрытый слой от выходного, тем более запутанной становится связь между его параметрами и стоимостью. В результате при движении от последнего скрытого слоя к первому градиент данного параметра по стоимости имеет склонность выравниваться, то есть он постепенно *исчезает*. По этой причине, как показано на рис. 8.8, чем дальше слой находится от выходного слоя, тем медленнее он обучается. Из-за этой проблемы *исчезающего градиента*, если бездумно добавлять в сеть все больше скрытых слоев, в конечном итоге скрытые слои, наиболее удаленные от выходного слоя, вообще не будут обучаться, нанося ущерб общей способности сети учиться аппроксимировать y по x.

ВЗРЫВНЫЕ ГРАДИЕНТЫ

Хотя и реже, чем в случае с исчезающими градиентами, некоторые сетевые архитектуры (например, рекуррентные сети, о которых рассказывается в главе 11) могут страдать проблемой *взрывных градиентов*. В этом случае градиент данного параметра относительно стоимости становится все *круче* при движении от последнего скрытого слоя к первому. Подобно исчезающим, взрывные градиенты могут ограничивать способность к обучению всей нейронной сети, насыщая нейроны экстремальными значениями.

ПАКЕТНАЯ НОРМАЛИЗАЦИЯ

Во время обучения нейронной сети распределение скрытых параметров в слое может постепенно меняться; это явление называется *внутренним сдвигом ковариации* (internal covariate shift). Фактически оно обусловлено самим смыслом обучения: мы должны изменить параметры, чтобы узнать что-то о базовых данных. Но из-за изменения распределения весов в слое входные данные для следующего слоя могут *сместиться* в сторону от идеального (то есть нормального, как на рис. 9.2) распределения. Решить эту проблему помогает прием *пакетной нормализации*[2]. Его суть заключается в следующем:

---

[1] Изменение прямо пропорционально отрицательной величине градиента с коэффициентом, равным скорости обучения η.

[2] *Ioffe S. & Szegedy C.* (2015). «Batch normalization: Accelerating deep network training by reducing internal covariate shift». *arXiv: 1502.03167*.

берутся выходные активации *a* предыдущего слоя, из них вычитается среднее по пакету, и результат делится на стандартное отклонение в пакете. Эта операция преобразует фактическое распределение активаций *a* в распределение со средним значением 0 и стандартным отклонением 1 (рис. 9.4). То есть если в предыдущем слое имеются какие-то экстремальные значения, они не вызовут взрывное увеличение или исчезновение градиентов в следующем слое. Кроме того, пакетная нормализация имеет следующие положительные эффекты:

- позволяет слоям учиться более независимо друг от друга, потому что большие значения в одном слое не оказывают чрезмерного влияния на вычисления в следующем;
- позволяет установить более высокую скорость обучения, потому что в нормированных активациях нет экстремальных значений, и тем самым обеспечивается более быстрое обучение;
- выходные данные слоя нормализуются по среднему значению и стандартному отклонению *пакета*, что добавляет элемент шума (особенно при небольших размерах пакетов), что, в свою очередь, способствует регуляризации. (Регуляризация рассматривается в следующем разделе, а пока просто отметим, что она помогает сети обобщать данные, которых она прежде не видела, что само по себе хорошо.)

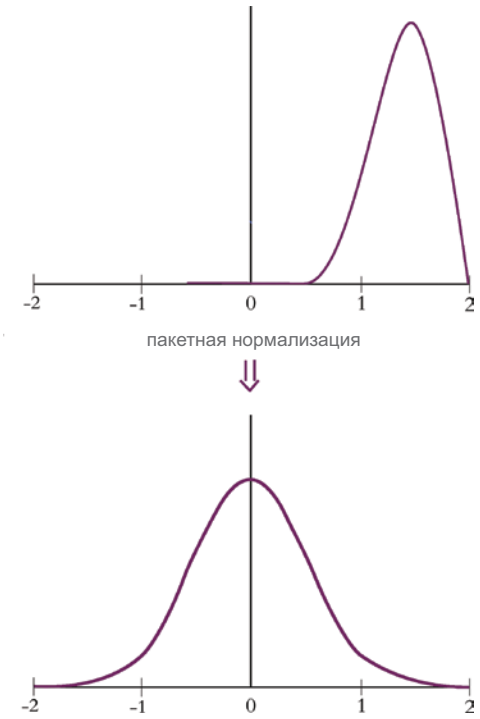


Рис. 9.4. Пакетная нормализация преобразует распределение активаций в стандартное нормальное распределение

Пакетная нормализация добавляет в каждый слой, к которому применяется, два дополнительных обучаемых параметра: γ (гамма) и β (бета). На последнем шаге пакетной нормализации выходы линейно преобразуются путем умножения на γ и прибавления β, где γ — это аналог стандартного отклонения, а β — среднего значения. (Возможно, вы заметили, что эта операция полностью обратна нормализации выходных значений!) Однако выходные значения первоначально нормализуются по среднему значению и стандартному отклонению *пакета*, тогда как γ и β изучаются алгоритмом SGD. Параметры γ и β в слое, подвергающемся пакетной нормализации, инициализируются как $\gamma = 1$ и $\beta = 0$, и, соответственно, в начале обучения это линейное преобразование не вносит никаких изменений; пакетная нормализация позволяет нормализовать выходы, как и предполагалось. Однако в процессе обучения сеть может определить, что *денормализация* активаций любого данного слоя могла бы способствовать снижению затрат. Поэтому, если пакетная нормализация оказывается вредной, сеть *учится* прекращать использовать ее в отдельных слоях. Фактически, так как γ и β являются непрерывными переменными, сеть может выбирать, *в какой степени* денормализовать результаты, в зависимости от того, что будет способствовать минимизации стоимости. Просто здорово!

ОБОБЩАЮЩАЯ СПОСОБНОСТЬ МОДЕЛИ (ПРЕДОТВРАЩЕНИЕ ПЕРЕОБУЧЕНИЯ)

В главе 8 упоминалось, что после обучения модели на протяжении определенного числа эпох стоимость, вычисляемая на проверочном наборе данных, которая существенно уменьшилась по сравнению с более ранними эпохами, может начать увеличиваться, даже при том что стоимость, вычисляемая на *обучающем* наборе данных, продолжает уменьшаться. Эта ситуация, когда стоимость на обучающих данных уменьшается, а на проверочных данных увеличивается, формально называется *переобучением*.

На рис. 9.5 показана графическая иллюстрация переобучения. Обратите внимание, что на всех графиках присутствуют одни и те же точки данных. Можно представить, что существует некоторое базовое распределение, которое описывают эти точки, и мы наблюдаем выборку из этого распределения. Наша цель — создать модель, которая объясняет отношения между x и y и которая, а это самое главное, пожалуй, *аппроксимирует* исходное распределение; то есть модель должна обобщать новые точки данных, полученные из распределения, а не просто моделировать имеющуюся выборку.

На первом графике (вверху слева на рис. 9.5) показана модель с одним параметром, которая ограничивается подгонкой прямой линии к данным[1]. Эта прямая

[1] Она моделирует линейную связь, простейшую форму регрессии между двумя переменными.

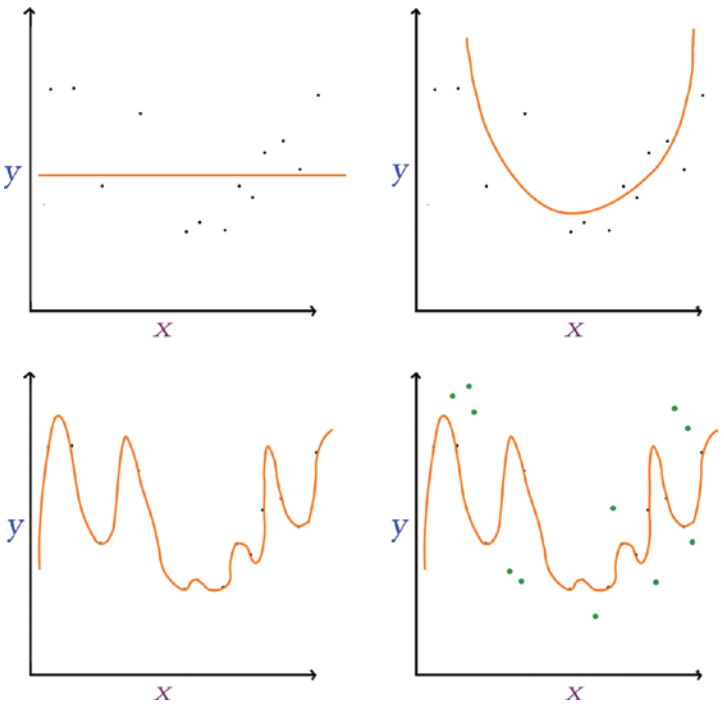


Рис. 9.5. Подгонка *y* по заданным *x* с использованием моделей с разным числом параметров. Вверху слева: модель с одним параметром плохо соответствует данным. Вверху справа: параболическая модель с двумя параметрами, которая в целом хорошо описывает связь между *x* и *y*. Внизу слева: модель с большим числом параметров точно описывает обучающие данные, но плохо обобщает новые точки (выделены зеленым цветом на графике внизу справа)

линия иллюстрирует *недообученную* модель: стоимость (вертикальные промежутки между линией и точками данных) высока, и такая модель плохо обобщает новые точки. Иначе говоря, линия *проходит мимо* большинства точек, потому что модель такого типа недостаточно сложна. На следующем графике (вверху справа) показана модель с двумя параметрами, имеющая форму параболы[1]. В этой параболической модели стоимость значительно ниже, чем в линейной, и, похоже, она способна обобщать новые данные — отлично!

На третьем графике (внизу слева на рис. 9.5) показана модель со слишком большим количеством параметров — оно превышает число имеющихся точек данных. Большое число параметров позволило уменьшить стоимость аппроксимации на обучающих данных до нуля: промежутки между кривой и точками данных отсутствуют. Однако на последнем графике (внизу справа) мы добавили новые точки данных из исходного распределения (отмечены зеленым цветом), которые модель не видела во время обучения, и поэтому они могут

[1] Вспомните квадратичную функцию из курса алгебры средней школы.

использоваться для ее проверки. Несмотря на полное отсутствие стоимости на обучающих данных, модель плохо аппроксимирует проверочные данные, что приводит к значительному увеличению стоимости на проверочных данных. В данном случае, дорогие друзья, модель с очень большим числом параметров оказалась *переобученной*: она идеально аппроксимирует обучающие данные, но не очень хорошо отражает отношения между x и y; она слишком подробно изучила особенности обучающих данных и поэтому плохо аппроксимирует данные, которые прежде не видела.

Как вы помните, тремя строками кода из листинга 5.6 мы создали неглубокую нейронную сеть с более чем 50 000 параметров (рис. 7.5). Учитывая эту простоту, неудивительно, что архитектуры глубокого обучения часто имеют миллионы параметров[1]. Использование моделей с таким большим числом параметров для обработки наборов данных, включающих лишь несколько тысяч обучающих образцов, легко может приводить к серьезному переобучению[2]. Можно ли извлечь выгоду от применения глубоких и сложных сетевых архитектур, не имея большого количества данных? К счастью, это возможно, потому что существуют методы, специально предназначенные для борьбы с переобучением. В этом разделе мы рассмотрим три таких метода: регуляризация L1/L2, прореживание и обогащение данных.

РЕГУЛЯРИЗАЦИЯ L1 И L2

В других областях машинного обучения, отличных от глубокого обучения, для ослабления эффекта переобучения широко используются методы *регуляризации L1 и L2*. Эти методы, которые также известны как *регрессия LASSO*[3] и *гребневая регрессия (ридж-регрессия)* соответственно, штрафуют модели за включение параметров, добавляя специальные параметры в функцию стоимости. Чем больше величина данного параметра, тем больший вклад он вносит в функцию стоимости. Соответственно, из модели исключаются параметры, которые не вносят заметного вклада в уменьшение разности между оценочными $\hat{y}$ и истинными значениями y. Проще говоря, отбрасываются избыточные параметры.

Методы регуляризации L1 и L2 отличаются тем, что в L1 прибавка к стоимости соответствует абсолютным величинам параметров, тогда как в L2 — их *квадратам*. В результате регуляризация L1 обычно приводит к включению в модель меньшего числа параметров с большими величинами, а регуляризация L2 — к включению большего числа параметров с малыми величинами.

---

[1] Более того, уже в главе 10 вам встретятся модели с десятками миллионов параметров.

[2] Чтобы показать, что число образцов должно намного превышать число параметров, можно использовать нотацию: $n \gg p$.

[3] Least Absolutely Shrinkage and Selection Operator — оператор наименьшего абсолютного сжатия и выбора.

ПРОРЕЖИВАНИЕ

Методы регуляризации L1 и L2 прекрасно справляются с ослаблением эффекта переобучения в моделях глубокого обучения, но на практике предпочтение обычно отдается методу регуляризации, характерному для нейронных сетей. Этот метод, называемый *прореживанием* (dropout), был разработан Джеффом Хинтоном (см. рис. 1.16) и его коллегами из Торонтского университета[1] и стал известен благодаря включению в их потрясающую архитектуру AlexNet (см. рис. 1.17).

Простая и мощная идея Хинтона и его коллег, направленная на предотвращение переобучения, представлена на рис. 9.6. Суть прореживания состоит в том, чтобы в каждом цикле *исключить* из обучения часть случайно выбранных нейронов. Для иллюстрации на рис. 9.6 показаны три цикла обучения[2]. В каждом цикле удаляется определенная доля случайно выбираемых нейронов в скрытых слоях. Из первого скрытого слоя сети удаляется треть нейронов (33.3%). Из второго — 50% нейронов. Теперь рассмотрим, как протекают три цикла обучения, изображенные на рис. 9.6:

1. На верхней диаграмме случайным образом исключается второй нейрон из первого скрытого слоя и первый нейрон из второго скрытого слоя.
2. На средней диаграмме исключаются первый нейрон из первого скрытого слоя и второй нейрон из второго. Сеть «не запоминает», какие нейроны исключались в предыдущих циклах обучения, и поэтому тот факт, что в первом и втором циклах из обучения были исключены разные нейроны, является чисто случайным совпадением.
3. На нижней диаграмме впервые из обучения исключается третий нейрон из первого скрытого слоя. Второй нейрон из второго скрытого слоя второй цикл подряд случайно исключается из обучения.

Вместо уменьшения величин параметров до нуля (как при пакетной нормализации), прореживание не ограничивает (напрямую) значение данного параметра. Тем не менее прореживание является эффективным методом регуляризации, потому что предотвращает чрезмерное влияние какого-то одного нейрона: благодаря исключению из процесса обучения какому-то очень специфическому аспекту обучающего набора данных сложнее создать слишком специфический путь прямого распространения через сеть, потому что в любом конкретном цикле обучения нейроны, формирующие этот путь, могут быть исключены. Соответственно, модель становится менее зависимой от конкретных характеристик данных при выборе прогноза.

---

[1] *Hinton G.* et al. (2012). «Improving neural networks by preventing co-adaptation of feature detectors». *arXiv:1207.0580*.

[2] Если вы запамятовали, что значит фраза «цикл обучения», обратитесь к рис. 8.5.

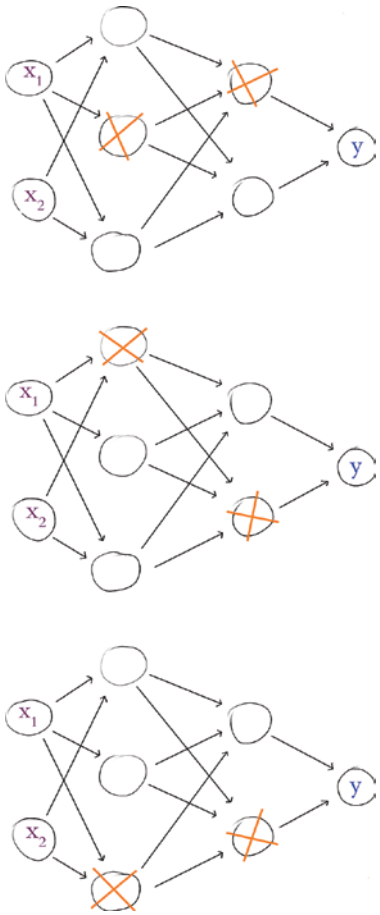


Рис. 9.6. Прореживание, прием для ослабления эффекта переобучения моделей, предполагает исключение случайно выбранных нейронов в каждом цикле обучения. Здесь показаны три цикла обучения с прореживанием

Перед проверкой модели нейронной сети, которая была обучена с использованием прореживания, или перед построением прогнозов с такой сетью нужно сделать один дополнительный шаг. При проверке или построении прогнозов желательно использовать всю мощь сети, то есть всю совокупность нейронов в ней. Загвоздка в том, что во время обучения использовалось только подмножество нейронов, через которые распространялись входные значения x и оценки $\hat{y}$. Если по наивности осуществить прямое распространение со *всеми* нейронами, это вызовет смещение оценок $\hat{y}$: с увеличением числа параметров итоговые значения после всех математических операций окажутся больше, чем ожидалось. Чтобы компенсировать влияние дополнительных нейронов, нужно соответствующим образом уменьшить параметры нейронов. Если, к примеру, во время обучения исключалась половина нейронов в скрытом слое, то перед проверкой или прогнозированием параметры слоя нужно умножить на 0.5. Второй пример: если при обучении из скрытого слоя исключалась треть ней-

ронов, то перед проверкой параметры нужно умножить на 0.667[1]. К счастью, библиотека Keras автоматически выполняет такую корректировку параметров. Однако при использовании других библиотек глубокого обучения (например, низкоуровневой библиотеки TensorFlow) вам может потребоваться проявить внимательность и не забывать выполнять эти корректировки самостоятельно.

Если вы знакомы с ансамблями статистических моделей (например, случайный лес из множества случайных деревьев решений), то для вас должно быть очевидно, что прореживание создает такой ансамбль. Во время каждого цикла обучения создается случайная подсеть, для которой происходит настройка значений параметров. Позднее, по завершении обучения, все эти подсети отражаются в значения параметров всей конечной сети. Соответственно, конечная сеть является агрегированным *ансамблем* составляющих его подсетей.

По аналогии со скоростью обучения и размером пакета (обсуждаются в главе 8), параметры сетевой архитектуры, управляющие прореживанием, являются гиперпараметрами. Вот наши практические рекомендации по выбору слоев и их количества для применения прореживания:

- Если ваша сеть страдает от эффекта переобучения (то есть стоимость на этапе проверки увеличивается, а на этапе обучения уменьшается), к такой сети желательно применить прием прореживания.
- Даже если эффект переобучения не проявляется, дополнительное прореживание может повысить точность проверки, особенно в более поздние эпохи обучения.
- Применение прореживания ко *всем* скрытым слоям в сети может быть излишним. Если сеть имеет достаточную глубину, прореживание можно применить только к последним слоям (самые ранние слои способны выявлять признаки, не вызывая переобучения). Чтобы убедиться в этом, можно для начала применить прореживание только к последнему скрытому слою и посмотреть, достаточно ли этого для предотвращения переобучения; если нет, добавьте прореживание к следующему самому глубокому слою и проверьте, что получилось, и т. д.
- Если сети не удается снизить стоимость на проверочных данных или получить такую же низкую стоимость, достигнутую при применении менее агрессивного прореживания, значит, вы чрезмерно увеличили долю исключаемых нейронов — уменьшите ее! Так же как в случае с другими гиперпараметрами, для прореживания существует своя оптимальная зона.
- В отношении *величины* доли исключаемых нейронов в данном слое каждая сеть ведет себя непохоже на другие, поэтому важно поэкспериментировать

[1] Иначе говоря, если вероятность, что данный нейрон останется включенным во время обучения, равна p, то параметры этого нейрона нужно умножить на p перед проверкой или прогнозированием.

и попробовать разные значения. По нашему опыту, исключение от 20% до 50% нейронов из скрытого слоя в приложениях компьютерного зрения обычно обеспечивает высочайшую точность на проверочных данных. Применительно к анализу естественного языка, где отдельные слова и фразы могут передавать особое значение, мы обнаружили, что оптимальной обычно является меньшая доля исключаемых нейронов — от 20% до 30%.

ОБОГАЩЕНИЕ ДАННЫХ

В дополнение к регуляризации параметров модели существует еще один метод предотвращения переобучения, заключающийся в увеличении размера обучающего набора данных. Если есть возможность недорого собрать дополнительные высококачественные обучающие данные для задачи, над которой вы работаете, сделайте это! Чем больше данных получит модель во время обучения, тем лучше она сможет обобщать проверочные данные, которые прежде не видела.

К сожалению, во многих случаях сбор дополнительных данных — несбыточная мечта. Однако есть другой путь — сгенерировать новые обучающие данные из существующих и таким способом искусственно расширить обучающий набор. Например, к коллекции цифр MNIST можно применить разные виды преобразований и получить обучающие выборки, состоящие из изображений рукописных цифр, например:

- перекос изображения;
- смазывание изображения;
- сдвиг изображения на несколько пикселов;
- добавление случайного шума в изображение;
- поворот изображения на небольшой угол.

В действительности, как показано на персональном веб-сайте Яна Лекуна (см. рис. 1.9), многие классификаторы цифр из коллекции MNIST, добивавшиеся рекордных результатов, использовали преимущество такого искусственного обогащения обучающего набора данных[1,2].

НЕОБЫЧНЫЕ ОПТИМИЗАТОРЫ

До сих пор в этой книге мы использовали только один алгоритм оптимизации: стохастический градиентный спуск. Несмотря на то что он показывает хорошие

[1] *yann.lecun.com/exdb/mnist*

[2] В главе 10 мы используем инструменты обогащения данных, имеющиеся в библиотеке Keras, на примере реальных изображений хот-догов.

результаты, исследователи разработали множество хитроумных способов, помогающих улучшить его.

МЕТОД МОМЕНТОВ

Первое усовершенствование стохастического градиентного спуска, которое мы рассмотрим, — *метод моментов*. Обратимся к аналогии для его описания: представьте, что сейчас зима, и наш бесстрашный трилобит спускается на лыжах вниз по снежному склону. Если на пути встречается локальный минимум (как на среднем кадре на рис. 8.7), инерция (момент) движения поможет трилобиту миновать этот минимум и продолжить спуск по склону. То есть градиенты, вычисленные на *предыдущих* шагах, повлияли на текущий шаг.

Момент в SGD вычисляется как скользящее среднее градиентов для каждого параметра и используется для корректировки весов на каждом шаге. При использовании метода моментов мы получаем дополнительный гиперпараметр β (бета) со значением в диапазоне от 0 до 1, который определяет, сколько предыдущих градиентов должно быть включено в скользящее среднее. Малые значения β позволяют вносить свой вклад более старым градиентам, что может быть нежелательно; едва ли трилобиту понравится, если при приближении к финишу его скорость будет определяться самой крутой частью склона в самом начале. Мы обычно используем большие значения β, и, как нам кажется, $\beta = 0.9$ выглядит вполне разумным значением по умолчанию.

МЕТОД НЕСТЕРОВА

Еще одна версия метода моментов называется *методом Нестерова*. В этом случае скользящее среднее градиентов *сначала* используется для корректировки весов и определения градиентов в точке, куда предстоит перейти; это эквивалентно быстрому взгляду на место, куда может привести инерция. Затем градиенты из этой точки используются для выполнения шага градиента *из текущей позиции*. Другими словами, трилобит внезапно осознает, насколько быстро он спускается, и начинает принимать эту скорость во внимание, чтобы предугадать, куда он может попасть, а затем корректирует направление еще до того, как попадет туда.

ADAGRAD

Оба метода моментов улучшают алгоритм стохастического градиентного спуска, но имеют свой недостаток — они используют одну и ту же скорость обучения η для всех параметров. Представьте, что у вас была бы возможность определять скорость обучения отдельно для каждого параметра, тогда вы могли бы замедлять или вообще останавливать обучение параметров, до-

стигших оптимального значения, и продолжать обучать параметры, далекие от оптимума. Вам повезло! Именно такую возможность предлагают другие оптимизаторы, которые мы обсудим в этом разделе: AdaGrad, AdaDelta, RMSProp и Adam.

Название *AdaGrad* — это сокращение от английского «adaptive gradient» (адаптивный градиент)[1]. В этом варианте каждый параметр имеет уникальную скорость обучения, которая меняется в зависимости от важности соответствующего признака. Этот метод особенно хорош при обучении на разреженных данных, когда некоторые признаки встречаются очень редко: если эти признаки появляются, иногда желательно более существенно скорректировать соответствующие им параметры. Индивидуализация достигается поддержкой матрицы с суммами квадратов прошлых градиентов для всех параметров и делением скоростей обучения на квадратные корни этих сумм. AdaGrad — это первый метод, где вводится параметр ϵ (эпсилон) — коэффициент сглаживания, помогающий избежать ошибок деления на ноль, и для него можно смело использовать значение по умолчанию $\epsilon = 1 \times 10^{-8}$.[2]

Большим преимуществом AdaGrad является почти полное отсутствие необходимости настраивать гиперпараметр скорости обучения η. В большинстве случаев можно просто установить значение по умолчанию $\eta = 0.01$ и забыть о нем. Существенный же недостаток AdaGrad — это деление скорости обучения на все большее и большее значение из-за разрастания матрицы прошлых градиентов, что в конечном итоге делает скорость обучения неоправданно малой и приводит к почти полной его остановке.

ADADELTA И RMSPROP

Метод *AdaDelta* устраняет недостаток AdaGrad, выражающийся в разрастании матрицы градиентов, поддерживая скользящее среднее по предыдущим градиентам по аналогии с методом моментов[3].

AdaDelta также исключает член η из уравнения и тем самым полностью избавляет от необходимости настраивать скорость обучения[4].

---

[1] *Duchi J.* et al. (2011). «Adaptive subgradient methods for online learning and stochastic optimization». *Journal of Machine Learning Research, 12*, 2121–59.

[2] AdaGrad, AdaDelta, RMSProp и Adam — все эти методы используют параметр ϵ с одной и той же целью, и во всех этих методах его можно заменить значением по умолчанию.

[3] *Zeiler M. D.* (2012). «ADADELTA: An adaptive learning rate method». *arXiv:1212.5701.*

[4] Это достигается с помощью остроумного математического трюка, который мы не будем здесь раскрывать. Однако вы можете заметить, что библиотеки Keras и TensorFlow все еще имеют параметр скорости обучения в своих реализациях AdaDelta. Мы рекомендуем устанавливать η равным 1, то есть не использовать масштабирование и, следовательно, не использовать функциональную скорость обучения, о которой вы узнали в этой книге.

Метод *RMSProp* (Root Mean Square Propagation — среднеквадратичное распространение) был разработан Джеффом Хинтоном (Geoff Hinton, см. рис. 1.16) примерно в то же время, что и AdaDelta[1]. Он работает почти так же, но сохраняет параметр скорости обучения η. Оба метода, RMSProp и AdaDelta, содержат дополнительный гиперпараметр ρ (ро) — скорость затухания, — которая является аналогом значения момента β и определяет размер окна для вычисления скользящего среднего. Для обоих оптимизаторов рекомендуется выбирать значение гиперпараметра $\rho = 0.95$ и $\eta = 0.001$ для RMSProp.

ADAM

Последним в этом разделе мы обсудим оптимизатор, который чаще других используется в книге. Метод *Adam* — сокращенно от «adaptive moment estimation» (адаптивная оценка моментов) — основывается на оптимизаторах, появившихся до него[2]. По сути, это тот же алгоритм RMSProp, но за двумя исключениями:

1. Для каждого параметра вычисляется дополнительное скользящее среднее прошлых градиентов (называется средним первым моментом градиента[3] или просто средним), которое используется для определения параметров в текущей точке вместо фактических градиентов.
2. Чтобы предотвратить смещение скользящих средних к нулю, в начале обучения используется остроумный трюк.

Метод Adam имеет два гиперпараметра, по одному для вычисления каждого из скользящих средних. Для них рекомендуется использовать следующие значения по умолчанию: $\beta_1 = 0.9$ и $\beta_2 = 0.999$. Для скорости обучения в методе Adam по умолчанию используется значение $\eta = 0.001$, и вы тоже можете использовать это значение в большинстве случаев.

Поскольку RMSProp, AdaDelta и Adam очень похожи, они взаимозаменяемы, хотя прием корректировки смещения в методе Adam может помочь на более поздних этапах обучения. Несмотря на популярность этих новомодных оптимизаторов, иногда все же лучше остановить выбор на простом методе моментов (или методе Нестерова), который в некоторых случаях работает лучше. По аналогии с другими аспектами моделей глубокого обучения вы можете экспериментировать с оптимизаторами и смотреть, какой лучше всего подходит для вашей конкретной модели и задачи.

---

[1] Этот оптимизатор до сих пор не опубликован. Впервые он был предложен в 6-й лекции курса «Нейронные сети для машинного обучения», который читает Хинтон (www.cs.toronto.edu/hinton/coursera/lecture6/lec6.pdf).

[2] *Kingma D. P. & Ba J.* (2014). «Adam: A method for stochastic optimization». *arXiv:1412.6980*.

[3] Другое скользящее среднее — среднее квадратов градиента — является вторым моментом градиента, или дисперсией.

ГЛУБОКАЯ НЕЙРОННАЯ СЕТЬ НА ОСНОВЕ KERAS

Теперь мы можем возвестить о достижении важного рубежа! Познакомившись с дополнительной теорией в этой главе, мы получили достаточный объем знаний для проектирования и обучения моделей. Если вы предпочитаете исследовать примеры в интерактивном режиме, откройте блокнот Jupyter *deep_net_in_keras.ipynb*. В сравнении с блокнотами, реализующими неглубокую модель и модель средней глубины (листинг 5.1), в этом блокноте появилась пара дополнительных зависимостей — для поддержки прореживания и нормализации, как показано в листинге 9.4.

Листинг 9.4. Дополнительные зависимости для глубокой сети на основе Keras

```
from keras.layers import Dropout
from keras.layers.normalization import BatchNormalization
```

Загрузка и предварительная обработка коллекции MNIST осуществляется точно так же, как прежде. Различия начинаются с определения архитектуры, как показано в листинге 9.5.

Листинг 9.5. Архитектура глубокой сети на основе Keras

```
model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))

model.add(Dense(10, activation='softmax'))
```

Как и прежде, мы создаем объект `Sequential` модели. Однако вслед за первым скрытым слоем мы также добавляем слой `BatchNormalization()`. При этом в модель добавляется не фактический слой с нейронами, а преобразование пакетной нормализации активаций из предыдущего слоя (первого скрытого слоя). Точно так же поверх второго скрытого слоя добавляем `BatchNormalization()`. Выходной слой сети идентичен тому, что использовался в сетях небольшой и средней глубины, но, чтобы создать настоящую *глубокую* нейронную сеть, мы добавляем третий скрытый слой. Как и предыдущие скрытые слои, третий скрытый слой состоит из `64` нормализованных нейронов `relu`, но дополнительно мы добавляем к этому последнему скрытому слою прореживание `Dropout`, отключающее одну пятую (`0.2`) нейронов в каждом цикле обучения.

Как показано в листинге 9.6, единственное, чем эта сеть отличается от предыдущей сети средней глубины, — вместо обычной оптимизации SGD используется оптимизатор Adam: `(optimizer = 'adam')`.

Листинг 9.6. Компиляция модели глубокой сети на основе Keras

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Обратите внимание, что оптимизатору Adam не требуется явно передавать значения гиперпараметров, потому что Keras автоматически подставляет достаточно оптимальные настройки по умолчанию, подробно описанные в предыдущем разделе. Библиотека Keras (и, соответственно, TensorFlow) предлагают реализации для всех оптимизаторов, рассмотренных выше, которые можно легко подставить вместо обычного SGD или Adam. Узнать, как это сделать, можно в документации к этим библиотекам, доступной в Интернете.

После вызова метода `fit()` модели[1] мы обнаруживаем, что изучение дополнительной теории в этой главе дало свои плоды: точность сети средней глубины на проверочных данных составила примерно 97.6%, а наша глубокая сеть достигла 97.87% после 15 эпох обучения (см. рис. 9.7), уменьшив на 11% и без того небольшой уровень ошибок. Чтобы выжать еще больше сока из лимона ошибок, нам понадобятся слои нейронов, специально предназначенные для компьютерного зрения, как те, что будут представлены в главе 10.

РЕГРЕССИЯ

В главе 4, при обсуждении задач обучения с учителем, упоминалось, что они делятся на задачи классификации и регрессии. Почти все модели в этой книге используются для классификации входных данных. Но в этом разделе мы отойдем от этого правила и посмотрим, как адаптировать модель нейронной сети для решения задач регрессии, то есть любых задач, в которых требуется предсказать некоторую непрерывную переменную. В качестве примеров регрессии можно назвать: прогнозирование цены акции, объема осадков в миллиметрах, которые могут выпасть завтра, или величины продаж конкретного продукта. В этом разделе мы используем нейронную сеть и классический набор данных для оценки стоимости жилья в пригороде Бостона, штат Массачусетс, в 1970-х годах.

В листинге 9.7 можно видеть, какие зависимости подключаются в нашем блокноте *regression_in_keras.ipynb*. Единственная незнакомая зависимость —

[1] Вызов `model.fit()` полностью совпадает с аналогичным вызовом в блокноте *intermediate_net_in_keras.ipynb* и в листинге 8.3.

это набор данных `boston_housing`, который для удобства поставляется вместе с библиотекой Keras.

```
Epoch 15/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.0288 - acc: 0.9906 - val_loss: 0.0865 - val_acc: 0.9787
Epoch 16/20
60000/60000 [==============================] - 1s 22us/step - loss: 0.0246 - acc: 0.9919 - val_loss: 0.0880 - val_acc: 0.9767
```

Рис. 9.7. Наша глубокая нейронная сеть достигла уровня точности 97.87% на проверочных данных после 15 эпох обучения, превысив точность архитектур с малой и средней глубиной. Из-за случайного характера инициализации начальных весов и обучения сети вы можете получить чуть более низкую или чуть более высокую точность

Листинг 9.7. Зависимости регрессионной модели

```
from keras.datasets import boston_housing
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers.normalization import BatchNormalization
```

Загрузка данных производится так же просто, как в случае с коллекцией цифр MNIST:

```
(X_train, y_train), (X_valid, y_valid) = boston_housing.load_data()
```

Если обратиться к параметру `shape` массивов `X_train` и `X_valid`, можно обнаружить, что у нас имеется 404 обучающих и 102 проверочных образца. В каждом образце — отдельном пригороде Бостона — задано 13 переменных, определяющих возраст здания, среднее число комнат, уровень преступности, соотношение числа студентов и преподавателей и т. д.[1] Медианная цена на жилье (в тысячах долларов) для каждого района определяется в переменных y. Например, в первом образце в обучающем наборе средняя цена дома составляет 15 200 долларов США[2].

Архитектура сети для предсказания цен на жилье показана в листинге 9.8.

Листинг 9.8. Архитектура регрессионной модели

```
model = Sequential()

model.add(Dense(32, input_dim=13, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(16, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))

model.add(Dense(1, activation='linear'))
```

[1] Узнать больше о данных можно в статье, где они были первоначально опубликованы: *Harrison D. & Rubinfeld D. L.* (1978). «Hedonic prices and the demand for clean air». Journal of Environmental Economics and Management, 5, 81–102.

[2] Элемент `y_train[0]` содержит число `15.2`.

При наличии всего 13 входных значений и нескольких сотен обучающих образцов использование глубокой нейронной сети с множеством нейронов в каждом слое не даст никакого преимущества, поэтому мы выбрали архитектуру с двумя скрытыми слоями, содержащими 32 и 16 нейронов соответственно. Мы также применили пакетную нормализацию и небольшое прореживание, чтобы избежать переобучения на конкретных обучающих образцах. Но *особенно важно*, что для выходного слоя мы выбрали линейную активацию (`activation='linear'`) — вариант, который часто используется, когда требуется предсказать непрерывную переменную, как в задачах регрессии. Линейная функция активации выводит z, поэтому оценка $\hat{y}$ сети может принимать любое числовое значение (представляющее, например, доллары, сантиметры), а не вероятность в диапазоне между 0 и 1 (как при использовании функций активации sigmoid и softmax).

При компиляции модели (см. листинг 9.9) мы использовали еще одну настройку, характерную для регрессионных моделей: применили функцию стоимости на основе среднеквадратичной ошибки (Mean Squared Error, MSE) вместо перекрестной энтропии (`loss = 'mean_squared_error'`). До сих пор мы использовали исключительно перекрестную энтропию, однако эта функция стоимости специально создавалась для задач классификации, в которых $\hat{y}$ является вероятностью. В задачах регрессии, где результат фактически не является вероятностью, обычно используется среднеквадратичная ошибка[1].

Листинг 9.9. Компиляция регрессионной модели

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

Возможно, вы заметили, что на этот раз мы опустили показатель точности. Мы сделали это преднамеренно: нет смысла вычислять точность, потому что этот показатель (процент правильно классифицированных образцов) не имеет отношения к непрерывным переменным[2].

Обучение модели (как показано в листинге 9.10) — это один из шагов, который выполняется иначе, чем в задачах классификации.

Листинг 9.10. Обучение регрессионной модели

```
model.fit(X_train, y_train,
          batch_size=8, epochs=32, verbose=1,
          validation_data=(X_valid, y_valid))
```

[1] Есть и другие функции стоимости, применимые к задачам регрессии, такие как средняя абсолютная ошибка (Mean Absolute Error, MAE) и потеря Хьюбера (Huber loss), но они не рассматриваются в этой книге. Среднеквадратичной ошибки вполне достаточно для наших целей.

[2] Также полезно помнить, что точность, как правило, используется только для нашего самоуспокоения в отношении качества работы моделей. Сама модель обучается на стоимости, а не на точности.

Мы установили продолжительность обучения равной 32 эпохам, потому что по нашему опыту работы с этой конкретной моделью знаем, что более продолжительное обучение не приводит к уменьшению потерь на проверочных данных. Мы не стали тратить время на оптимизацию гиперпараметра, определяющего размер пакета, но изменив его, вы можете получить небольшой прирост точности.

Экспериментируя с этой регрессионной моделью, мы достигли самых низких потерь на проверочных данных (25.7) в 22-й эпохе. К последней (32-й) эпохе величина потерь возросла до 56.5 (для сравнения, после первой эпохи обучения величина потерь на проверочных данных составила 56.6). В главе 11 мы покажем, как сохранять параметры модели после каждой эпохи обучения, чтобы позже можно было загрузить наиболее эффективную, но на данный момент мы имеем не самые лучшие параметры, установленные в последней эпохе. Но как бы то ни было, если вы захотите увидеть конкретные примеры прогнозов цен на жилье с учетом конкретных входных данных, запустите код, показанный в листинге 9.11[1].

Листинг 9.11. Прогнозирование медианной цены на жилье в конкретном районе Бостона

```
model.predict(np.reshape(X_valid[42], [1, 13]))
```

Этот код вернул нам прогнозную медианную цену на жилье ($\hat{y}$), равную 20 880 долларам США для 43-го района Бостона из проверочного набора. Фактическая медианная цена (y, которую можно получить, обратившись к `y_valid[42]`) равна 14 100 долларам США.

TENSORBOARD

При оценке качества модели между эпохами довольно утомительно и трудно читать отдельные результаты, как мы это делали после запуска кода в листинге 9.10, особенно если модель обучается на протяжении большого числа эпох. Чтобы облегчить свой труд, можно использовать TensorBoard (рис. 9.8) — удобный графический инструмент для:

- визуальной трассировки качества модели в реальном времени;
- оценки качества модели в прошлые эпохи;
- сравнения качества различных архитектур моделей и настроек гиперпараметров при обучении на одних и тех же данных.

[1] Обратите внимание, что нам пришлось использовать метод `reshape` из библиотеки NumPy, чтобы передать 13 переменных-предикторов из 43-го образца в виде массива векторов-строк (`[1, 13]`), а не в виде вектора-столбца.

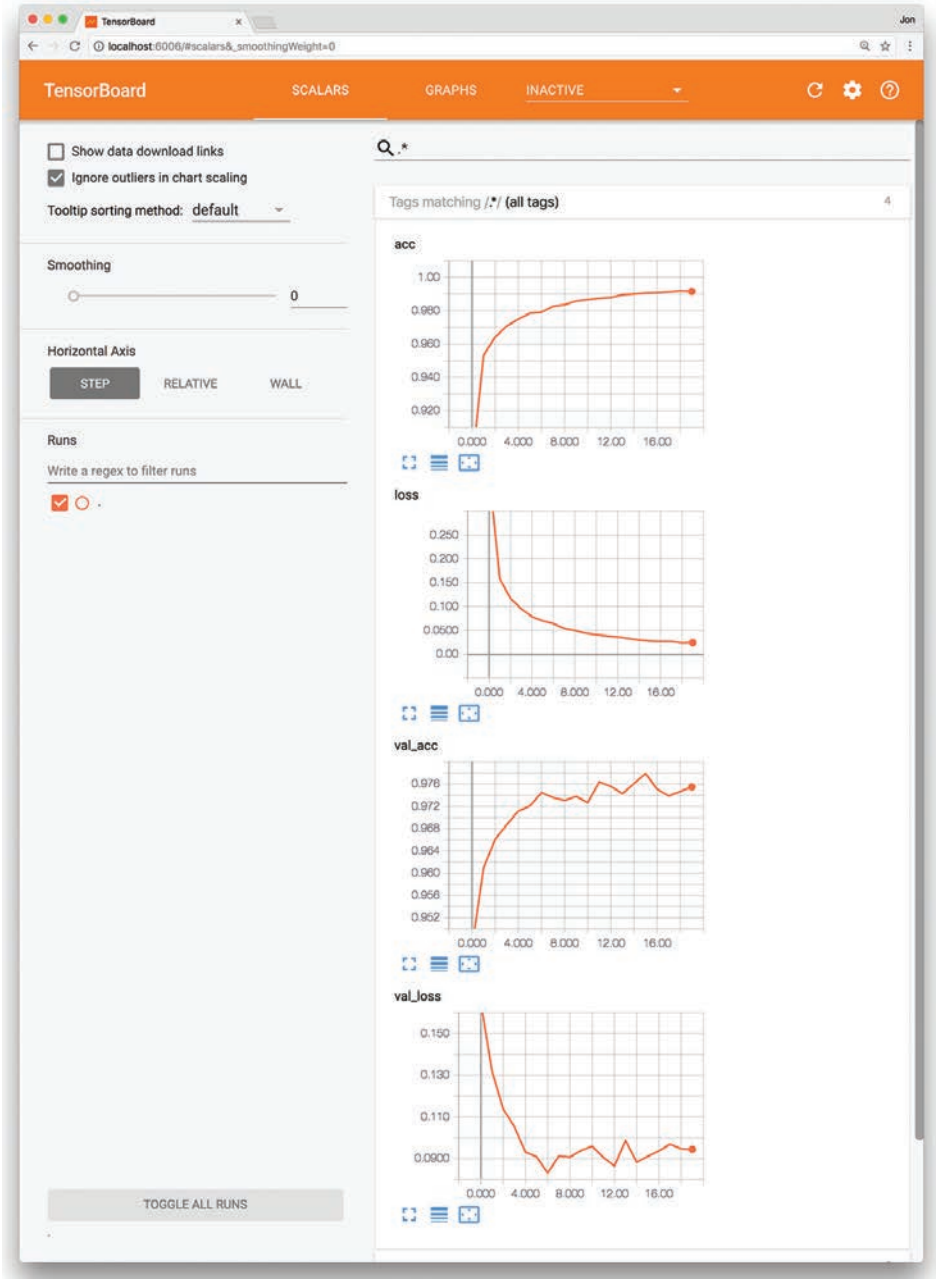


Рис. 9.8. Панель инструментов TensorBoard позволяет визуально следить за изменением стоимости (loss) и точности (acc) модели на обучающих и проверочных (val) данных на всем протяжении обучения

TensorBoard поставляется вместе с библиотекой TensorFlow, а инструкции по его использованию и настройке доступны на сайте TensorFlow[1]. Обычно это не вызывает проблем. Например, вот процедура, адаптирующая наш блокнот *deep_net_in_keras.ipynb* для использования TensorBoard в Unix-подобной операционной системе, включая macOS.

1. Как показано в листинге 9.12, измените код на Python следующим образом[2]:
   а) импортируйте класс `TensorBoard` из пакета `keras.callbacks`;
   б) создайте экземпляр `TensorBoard` (мы дали ему имя `tensorboard`) и укажите новое и уникальное имя каталога (например, `deep-net`), куда TensorBoard будет сохранять данные, относящиеся к конкретному сеансу обучения модели:

      ```
      tensorboard = TensorBoard(log_dir='logs/deep-net')
      ```

   в) передайте объект TensorBoard методу `fit()` в параметре `callback`:

      ```
      callbacks = [tensorboard]
      ```

2. В окне терминала выполните следующую команду[3]:

   ```
   tensorboard --logdir='logs/deep-net' --port 6006
   ```

3. Откройте в веб-браузере страницу `localhost:6006`.

Листинг 9.12. Использование TensorBoard для трассировки процесса обучения модели

```
from keras.callbacks import TensorBoard
tensorboard = TensorBoard('logs/deep-net')
model.fit(X_train, y_train,
          batch_size=128, epochs=20,
          verbose=1,
          validation_data=(X_valid, y_valid),
          callbacks=[tensorboard])
```

Выполнив эти или аналогичные шаги, в зависимости от конкретной операционной системы вы должны увидеть в окне веб-браузера панель инструментов, изображенную на рис. 9.8. С ее помощью вы сможете в масштабе реального времени наблюдать, как изменяются стоимость и точность любой конкретной модели от эпохи к эпохе на наборах обучающих и проверочных данных. Этот

---

[1] tensorflow.org/guide/summaries_and_tensorboard

[2] Эти изменения также отражены в нашем блокноте *deep_net_in_keras_with_tensorboard.ipynb*.

[3] Мы указали тот же каталог для сохранения журналов, который задан в настройках объекта TensorBoard на шаге 1b. Поскольку мы указали относительный путь, а не абсолютный, команда `tensorboard` должна запускаться в том же каталоге, где находится блокнот *deep_net_in_keras_with_tensorboard.ipynb*.

способ наблюдения за изменением качества модели является одним из основных применений TensorBoard, хотя интерфейс панели мониторинга поддерживает также множество других функций, таких как визуальное отображение графа нейронной сети и распределение весов в модели. Узнать об этих дополнительных функциях можно в документации к TensorBoard и изучив интерфейс самостоятельно.

ИТОГИ

В этой главе мы обсудили подводные камни, наиболее часто встречающиеся в моделировании нейронных сетей, и рассмотрели стратегии минимизации их влияния на качество модели. В завершение главы мы применили все теоретические основы, изученные к этому моменту, чтобы построить первую настоящую сеть глубокого обучения, которая обеспечила еще более высокую точность классификации рукописных цифр из коллекции MNIST. В общем случае такие глубокие и полносвязанные нейронные сети обеспечивают хорошую аппроксимацию любых выходных данных y по некоторым входным данным x, но иногда они оказываются не самым эффективным средством для создания специализированных моделей. Далее, в части III, мы познакомимся с новыми слоями нейронов и подходами к глубокому обучению, которые позволяют добиться превосходных результатов в конкретных задачах, включая компьютерное зрение, обработку естественного языка, создание образцов художественного искусства и игры.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым.

- параметры:
  - вес w;
  - смещение b;
- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
  - Linear;
- входной слой;
- скрытый слой;

- выходной слой;
- типы слоев:
  - плотный (полносвязанный);
  - softmax;
- функции стоимости (потерь):
  - квадратичная (среднеквадратичная ошибка);
  - перекрестная энтропия;
- прямое распространение;
- обратное распространение;
- нестабильность градиентов (в частности затухание);
- метод Глоро инициализации весов;
- пакетная нормализация;
- прореживание;
- оптимизаторы:
  - стохастический градиентный спуск;
  - Adam;
- гиперпараметры оптимизаторов:
  - скорость обучения η;
  - размер пакета.

III

ИНТЕРАКТИВНЫЕ ПРИЛОЖЕНИЯ ГЛУБОКОГО ОБУЧЕНИЯ

10

КОМПЬЮТЕРНОЕ ЗРЕНИЕ

Добро пожаловать в третью часть книги! Выше вы познакомились с общим обзором сфер применения глубокого обучения (часть I) и с основополагающей теорией (часть II). Теперь вы обладаете всеми необходимыми знаниями, чтобы приступить к чтению специализированных глав практической направленности, насыщенных примерами программного кода. Например, в этой главе вы познакомитесь со сверточными нейронными сетями и научитесь применять их для решения задач компьютерного зрения. В последующих главах в этой части книги мы также рассмотрим следующие практические примеры:

- в главе 11 — применение рекуррентных нейронных сетей для обработки естественного языка;
- в главе 12 — использование генеративно-состязательных сетей для создания художественных произведений;
- в главе 13 — реализация принятия последовательных решений в сложных меняющихся условиях с использованием глубокого обучения с подкреплением.

СВЕРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ

Сверточные нейронные сети (Convolutional Neural Network, ConvNet, CNN) — это искусственные нейронные сети, имеющие один или несколько *сверточных слоев*. Слои этого типа позволяют моделям глубокого обучения эффективно обрабатывать пространственные структуры. Как вы увидите далее в этой главе, это свойство делает сверточные слои особенно эффективными в приложениях компьютерного зрения.

ДВУМЕРНАЯ СТРУКТУРА ВИЗУАЛЬНЫХ ИЗОБРАЖЕНИЙ

В наших предыдущих примерах реализации распознавания рукописных цифр из коллекции MNIST мы преобразовывали изображения в одномерные массивы чисел, чтобы затем передать их в полносвязанный скрытый слой. В частности, мы брали исходные черно-белые изображения с размерами 28 × 28 пикселов и преобразовывали их в одномерные массивы по 784 элемента[1]. Этот шаг был необходим из-за использования полносвязанной сети — мы должны были преобразовать двумерные изображения в плоские массивы по 784 пиксела, чтобы ввести их в нейроны первого скрытого слоя. Однако такое преобразование двумерного изображения в одномерный массив влечет существенную потерю информации о структуре изображения. Рисуя цифру на бумаге, вы не представляете ее как непрерывную линейную последовательность пикселов от левого верхнего угла до правого нижнего. Если, например, вывести цифру из коллекции MNIST в виде линейной последовательности 784 черно-белых пикселов, мы готовы поспорить, что вы не сможете распознать ее. Люди воспринимают визуальную информацию в двумерной форме[2], и наша способность распознавать изображения неразрывно связана с пространственными отношениями между воспринимаемыми формами и цветами.

ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ

Кроме потери информации о двумерной структуре при преобразовании изображения в одномерный массив, вторым важным аспектом, связанным с передачей изображений в полносвязанную сеть, является вычислительная сложность. Изображения MNIST очень маленькие, они содержат всего 28 × 28 пикселов с одним *цветовым каналом* (цифры в коллекции MNIST являются монохроматическими; для отображения полноцветных изображений необходимо как минимум три канала — обычно красный, зеленый и синий). Для передачи информации об изображении из коллекции MNIST в полносвязанный слой каждый его нейрон должен иметь 785 параметров: 784 веса, соответствующих каждому пикселу, плюс смещение нейрона. Однако уже для передачи изображения умеренного размера — скажем, полноцветного, в формате RGB[3], размером 200 × 200 пикселов — нейроны должны иметь намного больше параметров. В этом случае у нас было бы три цветовых канала, в каждом по 40 000 пикселов, что соответствует 120 001 параметру на нейрон в полносвязанном слое[4]. При весьма скромном количестве нейронов в полносвязанном слое — скажем, 64 — это соответствует

---

[1] Напомним, что значения пикселов были разделены на 255, чтобы привести их к диапазону [0 : 1].

[2] Конечно же, в трехмерной, но такие детали не важны для предмета нашего обсуждения.

[3] Для полноцветного изображения необходимы красный (Red), зеленый (Green) и синий (Blue) каналы.

[4] 200 × 200 пикселов × 3 цветовых канала + 1 смещение = 120 001 параметр.

почти 8 миллионам параметров только в первом скрытом слое сети[1]. Более того, изображение размером *всего* 200 × 200 пикселов может быть получено камерой с разрешающей способностью 0.4 Мп[2], тогда как большинство современных смартфонов имеют камеры с разрешением 12 Мп и даже больше. Как правило, для успешного решения задач компьютерного зрения не требуется использовать изображения с высоким разрешением, но суть, мы надеемся, вам понятна: изображения могут иметь огромное число точек данных и для обработки их с использованием полносвязанных слоев могут потребоваться гигантские вычислительные мощности.

СВЕРТОЧНЫЕ СЛОИ

Сверточные слои состоят из наборов *ядер*, которые также называются *фильтрами*. Каждое из ядер представляет небольшое окно, сканирующее изображение (выражаясь техническим языком, *сворачивает*), из верхнего левого угла в правый нижний (см. иллюстрацию *операции свертки* на рис. 10.1).

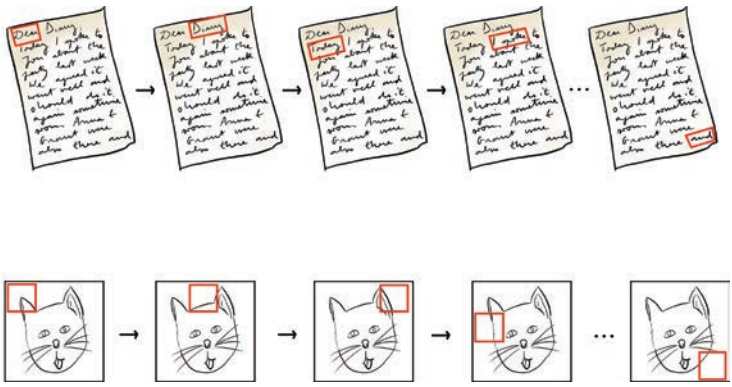


Рис. 10.1. Чтение страниц книги мы начинаем с левого верхнего угла и заканчиваем в правом нижнем. Каждый раз, достигая конца строки, мы переходим к началу следующей. Так мы рано или поздно достигаем правого нижнего угла, прочитав все слова на странице. Ядро в сверточном слое тоже начинает сканирование в левом верхнем углу изображения, перемещаясь слева направо, ряд за рядом, пока наконец не достигнет правого нижнего угла, просканировав все пикселы изображения

Ядра состоят из весов, которые, как в полносвязанных слоях, корректируются в ходе обратного распространения. Они могут иметь разный размер, но чаще используются ядра 3 × 3, и мы также используем этот размер в примерах в этой главе[3]. Для одноцветных цифр MNIST такое окно размером 3 × 3 пиксела будет

[1] 64 нейрона × 120 001 параметр на нейрон = 7 680 064 параметров.

[2] Мегапикселов.

[3] Другой типичный размер — 5 × 5. Ядра большего размера редко используются на практике.

состоять из 3 × 3 × 1 весов — девяти весов и в целом 10 параметров (подобно искусственным нейронам в полносвязанном слое, каждый сверточный фильтр имеет член смещения b). Для сравнения, если бы мы работали с полноцветными RGB-изображениями, ядро, охватывающее такое же число пикселов, имело бы в три раза больше весов — 3 × 3 × 3, всего 27 весов и 28 параметров.

Как показано на рис. 10.1, ядро последовательно занимает дискретные позиции на изображении по мере его свертывания. Исходя из размера ядра 3 × 3, которое мы условились использовать в этом объяснении, в ходе прямого распространения для каждой позиции, в которой оказывается ядро, вычисляется многомерный вариант «самого важного уравнения в этой книге» — $w \cdot x + b$ (представленного на рис. 6.7). В соответствии с размером окна 3 × 3 и ядра 3 × 3, как показано на рис. 10.2, на основе входных данных x и весовых коэффициентов w определяется взвешенная сумма $w \cdot x$, в которой произведения вычисляются поэлементно, по вертикали и горизонтали. Для более полного понимания полезно мысленно наложить ядро на значения пикселов. Вот как выглядят сами вычисления:

$$\begin{aligned} w \cdot x = &.01 \times .53 + .09 \times .34 + .22 \times .06 + \\ &+ -1.36 \times .37 + .34 \times .82 + -1.59 \times .01 + \\ &+ .13 \times .62 + -.69 \times .91 + 1.02 \times .34 = \\ &= -0.3917. \end{aligned} \tag{10.1}$$

.01	.09	.22
-1.36	.34	-1.59
.13	-.69	1.02

веса ядра

•

.53	.34	.06
.37	.82	.01
.62	.91	.34

пикселы на входе

Рис. 10.2. Ядро с размером 3 × 3 и окно 3 × 3 пиксела

Далее, используя уравнение 7.1, добавим некоторое смещение, член b (например, –0.19) и получим z:

$$\begin{aligned} z &= w \cdot x + b = \\ &= -0.39 + b = \\ &= -0.39 + 0.20 = \\ &= -0.19. \end{aligned} \tag{10.2}$$

Получив z, можно вычислить значение активации a, передав z в выбранную функцию активации, например tanh или ReLU.

Обратите внимание, что основная процедура вычислений осталась прежней, как при расчетах для искусственных нейронов в главах 6 и 7. Ядра свертки имеют **веса**, **входные данные** и **смещение**; взвешенная сумма для них вычисляется с использованием все того же самого важного уравнения, а полученный результат z передается в некоторую нелинейную функцию для получения **активации**. Единственное, что изменилось, — теперь у нас нет весов для *каждого* входного значения, а есть дискретное ядро с 3×3 весами. Эти веса не меняются в процессе свертывания — они остаются *общими* для всех входных значений. В результате сверточный слой может иметь на несколько порядков меньше весов, чем полносвязанный слой. Другая важная особенность: подобно входным данным, выходы ядра (все активации) также имеют форму двумерного массива. Мы еще вернемся к этому чуть ниже, но сначала...

МНОЖЕСТВО ФИЛЬТРОВ

Как правило, в сверточном слое имеется несколько фильтров. Каждый из них позволяет сети выявить уникальное представление данных на конкретном уровне. Например, подобно простым нейронам биологической системы зрения в экспериментах Хьюбела и Визеля (см. рис. 1.5), если первый скрытый слой в нашей сети является сверточным, он может содержать ядро, сильнее всего реагирующее на вертикальные линии. То есть всякий раз, обнаруживая вертикальную линию в процессе свертывания входного изображения, он выдает большое значение активации (a). Дополнительные ядра в этом слое могут учиться представлять другие простые пространственные признаки, такие как горизонтальные линии и переходы цветов (например, см. кадр слева внизу на рис. 1.17). Именно поэтому ядра стали называть *фильтрами*; они сканируют изображение и отфильтровывают местоположение определенных объектов, генерируя высокие активации, встретив шаблон, форму и/или цвет, для обнаружения которых они специально настроены. Можно сказать, что они функционируют как маркеры, генерируя двумерный массив активаций, указывающих, *где* в исходном изображении находится признак, распознаваемый этим фильтром. По этой причине вывод ядра называется *картой активации*.

Аналогично иерархическим представлениям в биологической системе зрения (см. рис. 1.6), последующие сверточные слои получают эти карты активации в качестве входных данных. По мере того как сеть становится глубже, фильтры в слоях реагируют на все более сложные комбинации этих простых признаков, обучаясь представлять все более абстрактные пространственные структуры и в конечном итоге выстраивая иерархию от простых линий и цветов до сложных текстур и форм (см. кадры внизу на рис. 1.17). В результате более глубокие слои сети способны распознавать целые объекты и даже, например, отличать немецкого дога от йоркширского терьера.

Количество фильтров в слое, как и количество нейронов в полносвязанном слое, является гиперпараметром, который мы должны настраивать сами. Подобно другим гиперпараметрам, описанным в этой книге, количество фильтров имеет оптимальное значение. Вот наши рекомендации по выбору числа фильтров для некоторой конкретной задачи:

- Большое количество ядер упрощает идентификацию сложных признаков, поэтому учитывайте сложность данных в решаемой задаче. Конечно, чем больше ядер, тем выше требования к вычислительным ресурсам.
- Если сеть имеет несколько сверточных слоев, оптимальное количество ядер для каждого слоя может значительно отличаться. Имейте в виду, что ранние слои идентифицируют простые признаки, тогда как более поздние — сложные комбинации этих простых признаков. Помните об этом, конструируя свою сеть. Как будет показано далее в этой главе, когда мы перейдем к примерам реализации CNN, обычно в задачах компьютерного зрения в более поздних сверточных слоях имеется гораздо больше ядер, чем в ранних.
- Как всегда, старайтесь свести к минимуму сложность вычислений: лучше использовать минимальное количество ядер, обеспечивающих минимальную стоимость на проверочных данных. Если удвоение количества ядер (скажем, с 32 до 64 и затем до 128) в данном слое значительно снижает стоимость на проверочных данных, используйте большее значение. Если уменьшение числа ядер (скажем, с 32 до 16 и затем до 8) в данном слое не увеличивает стоимость на проверочных данных, используйте меньшее значение.

ПРИМЕР СВЕРТОЧНОЙ СЕТИ

Сверточные слои являются нетривиальным отступлением от более простых полносвязанных слоев, о которых рассказывалось во второй части книги, поэтому, чтобы помочь вам понять, как значения пикселов и веса объединяются для создания карт признаков, на рис. 10.3—10.5 мы воспроизвели подробный вымышленный пример с математическими вычислениями. Для начала представьте, что выполняется свертка изображения в формате RGB размером 3 × 3 пиксела. В Python эти данные хранятся в виде массива `[3, 3, 3]`, как показано в верхней части на рис. 10.3[1].

В середине показаны массивы 3 × 3, соответствующие трем каналам цвета: красному, зеленому и синему. Обратите внимание, что изображение дополнено нулями со всех четырех сторон. Об этом дополнении мы поговорим чуть ниже, а пока просто помните, что дополнение используется, чтобы гарантировать совпадение размеров карты признаков и входных данных. Ниже массивов со значениями пикселов показаны весовые матрицы для всех трех каналов. Мы выбрали размер ядра 3 × 3, и, учитывая, что входное изображение имеет три

[1] Мы понимаем, что RGB-изображение дерева состоит из более чем девяти пикселов, мы просто старались сымитировать цветное изображение размером 3 × 3.

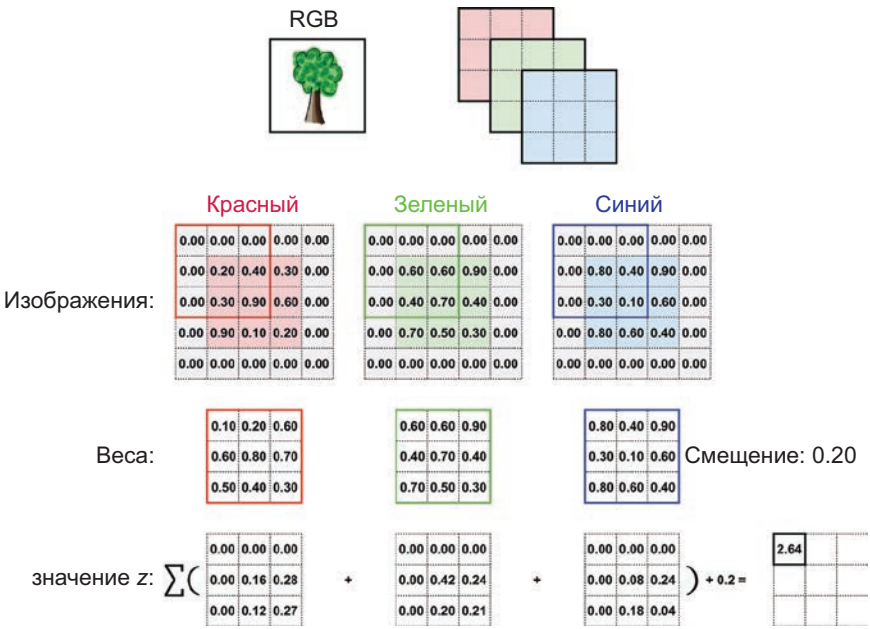


Рис. 10.3. Эта схематическая диаграмма демонстрирует, как сверточный слой вычисляет значения активаций в карте признаков

канала, матрица весов будет представлена массивом `[3, 3, 3]`, содержимое которого здесь показано отдельно для каждого канала. Для смещения выбрано значение 0.2. Текущая позиция фильтра обведена рамкой на изображении каждого массива со значениями пикселов, а в правом нижнем углу приводится значение *z* (вычисляется в соответствии с уравнением 10.1 как взвешенная сумма для всех трех цветовых каналов, к которой прибавляется смещение, согласно уравнению 10.2). Наконец, все значения *z* суммируются, и получается первый элемент в карте признаков, показанной в правом нижнем углу.

Далее, на рис. 10.4, массивы пикселов показаны с фильтром в следующей позиции, смещенной на один пиксел вправо. Значение *z* снова вычисляется в соответствии с уравнениями 10.1 и 10.2 и заполняет вторую позицию в карте активаций, как показано справа внизу на рис. 10.3.

Этот процесс повторяется для всех возможных позиций фильтра, и для каждой из этих девяти позиций вычисляется значение *z*, как показано справа внизу на рис. 10.5. Чтобы преобразовать карту 3 × 3 значений *z* в соответствующую карту 3 × 3 активаций, каждое значение *z* передается в функцию активации, такую как ReLU. Поскольку сверточный слой почти всегда имеет несколько фильтров, каждый из которых создает свою двумерную карту активаций, конечная карта имеет дополнительное измерение — *глубину*, — подобно той, которая определяется числом каналов в RGB-изображении. Каждый из этих ядерных

Рис. 10.4. Продолжение примера свертки, теперь вычисляется активация для следующей позиции фильтра

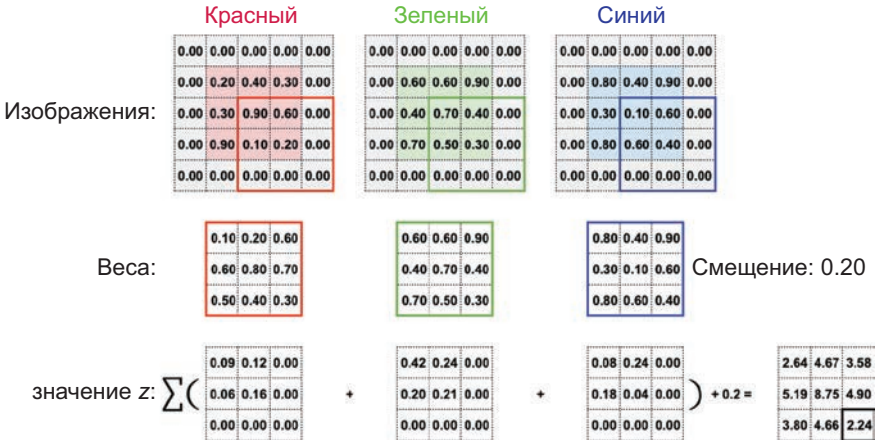


Рис. 10.5. Наконец, вычислением активаций для последней позиции фильтра завершается составление карты активаций

«каналов» в карте активаций представляет признак, на распознавание которого настроено конкретное ядро, например ребро (прямая линия) с определенной ориентацией[1]. На рис. 10.6 показано, как из исходного изображения путем вычисления активаций *a* строится трехмерная карта активаций. Сверточный слой, создавший карту активаций на рис. 10.6, имеет 16 ядер, то есть он генерирует карту активаций с глубиной 16 каналов (в дальнейшем мы будем называть их *срезами*).

[1] На рис. 1.17 показаны реальные примеры признаков, на выявлении которых специализируются отдельные ядра. Например, в первом сверточном слое большинство ядер специализируется на обнаружении ребер с определенной ориентацией.

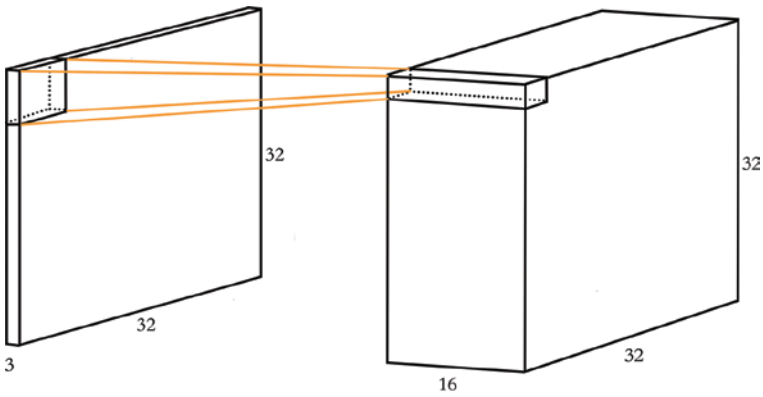


Рис. 10.6. Графическое представление входного массива (слева; здесь представлено RGB-изображение размером 32 × 32 и ядром, находящимся в первой позиции, то есть в левом верхнем углу) и карты активаций (справа). Всего имеется 16 ядер, соответственно, карта активаций имеет глубину 16. Каждой возможной позиции ядра во входном изображении соответствует одно значение в итоговой карте активаций

На рис. 10.6 окно фильтров расположено в верхнем левом углу исходного изображения. Ему соответствуют 16 значений в верхнем левом углу карты активаций: по одной активации *a* для каждого из 16 ядер. В процессе свертывания перемещением фильтра через все возможные позиции во входном изображении, слева направо и сверху вниз, заполняются все элементы карты активаций[1]. Если первый из 16 фильтров настроен на выявление вертикальных линий, тогда первый срез карты активаций выделит все области на изображении, содержащие вертикальные линии. Если второй фильтр настроен на выявление *горизонтальных* линий, тогда второй срез карты активаций выделит области, содержащие горизонтальные линии. Таким образом, все 16 фильтров в карте активаций могут вместе представлять пространственное расположение 16 различных пространственных признаков[2].

[1] Обратите внимание, что для любого изображения — монохроматического (с одним каналом цвета) или полноцветного (с тремя каналами) — каждое сверточное ядро дает только *одну* карту активаций. Если имеется только один канал цвета, взвешенная сумма входов вычисляется для этого одного канала, как в уравнении 10.1. Если имеется три канала цвета, общая взвешенная сумма входов вычисляется по всем трем каналам, как показано на рис. 10.3, 10.4 и 10.5. В любом случае (после добавления смещения ядра и передачи полученного значения *z* в функцию активации) для каждой позиции, над которой каждое ядро выполняет операцию свертки, создается только одно значение активации.

[2] Если вам интересно увидеть интерактивную демонстрацию вычислений в сверточном фильтре, мы настоятельно рекомендуем прочитать статью «Convolution Demo», которую написал Андрей Карпатый (Andrej Karpathy, рис. 14.6). Она доступна по адресу bit.ly/CNNdemo.

На этом этапе многие студенты, изучающие глубокое машинное обучение, часто задаются вопросом: откуда берутся веса для данного сверточного ядра. В наших примерах в этом разделе все значения параметров были выдуманы. Однако на практике веса и смещения ядра инициализируются случайными значениями (как рассказывалось в главе 9), а затем корректируются методом обратного распространения, подобно весам и смещениям в полносвязанных слоях. Как предполагает иерархическая абстракция, представленная в главе 1, самые ранние сверточные слои в глубокой сверточной сети (CNN) обычно настраиваются на выявление простых признаков, таких как прямые линии с разной ориентацией, а более глубокие слои специализируются на выявлении более сложных признаков, таких как лица, часы или собаки. Четырехминутный видеоролик, созданный Джейсоном Йосински (Jason Yosinski) и его коллегами (доступен на *bit.ly/DeepViz*), наглядно демонстрирует специализацию сверточных ядер в зависимости от глубины слоя в сети[1]. Мы настоятельно рекомендуем посмотреть его.

Теперь, когда мы описали общие принципы работы сверточных слоев в глубоком обучении, самое время рассмотреть основные их свойства:

- они позволяют моделям глубокого обучения учиться выявлять разные признаки в той или иной позиции, одно ядро может выявлять свой уникальный признак в любом месте во входных данных;
- они сохраняют двумерную структуру изображений, позволяя идентифицировать признаки в пространственном контексте;
- они значительно сокращают количество параметров, необходимых для моделирования изображений, обеспечивая более высокую эффективность вычислений;
- в конечном счете они обеспечивают более точное решение задач компьютерного зрения (например, классификация изображений).

ГИПЕРПАРАМЕТРЫ СВЕРТОЧНЫХ ФИЛЬТРОВ

Сверточные слои по своей природе *не являются* полносвязанными. То есть не существует весов, отображающих каждый пиксел в каждый нейрон в первом скрытом слое. Вместо этого имеется несколько гиперпараметров, которые определяют количество весов и смещений в данном сверточном слое. В том числе:

- размер ядра;
- длина шага;
- дополнение.

[1] *Yosinski J.* et al. (2015). «Understanding neural networks through deep visualization». Proceedings of the International Conference on Machine Learning.

Размер ядра

Во всех примерах, рассматривавшихся выше в этой главе, использовался *размер ядра* (также известный как *размер фильтра* или *рецептивное поле*[1]) 3 пиксела в ширину и 3 пиксела в высоту. Это типичный размер, обеспечивающий высокую эффективность в широком спектре задач компьютерного зрения с использованием современных сверточных архитектур. Размер ядра 5 × 5 тоже пользуется популярностью, а 7 × 7 обычно считается максимальным. Если ядро слишком велико по отношению к изображению, в рецептивном поле окажется слишком много конкурирующих признаков и сверточному слою будет сложнее обучаться, а если оно будет слишком мало (например, 2 × 2), то не сможет выявлять какие-либо структуры, что тоже плохо.

Длина шага

Под *длиной шага* подразумевается размер шага перемещения ядра по изображению. В нашем примере со сверточным слоем (рис. 10.3–10.5) мы использовали длину шага в 1 пиксел, которая часто применяется на практике. Также нередко используется длина шага в 2 пиксела и реже — в 3 пиксела. Выбор большей длины шага, скорее всего, окажется неоптимальным, потому что ядро будет пропускать области изображения, имеющие значение для модели. С другой стороны, увеличение шага ведет к увеличению скорости, потому что требуется меньше вычислений. Как и всегда в глубоком обучении, важно найти правильный баланс между точностью и скоростью. Мы рекомендуем использовать длину шага 1 или 2 и избегать длины шага больше 3.

Дополнение

Следующий гиперпараметр — *дополнение*. Он вместе с длиной шага играет важную роль в упорядочении вычислений, производимых в сверточном слое. Предположим, у вас есть изображение 28 × 28 из коллекции MNIST и ядро 5 × 5. При длине шага 1 ядро должно миновать 24 × 24 позиций, прежде чем пересечет изображение из края в край, поэтому карта активаций в этом слое получится немного меньше входного изображения. Чтобы получить карту активаций, точно соответствующую размеру входного изображения, можно просто дополнить изображение нулями по краям (как на рис. 10.3, 10.4 и 10.5). В случае изображения 28 × 28 и ядра 5 × 5, дополнение двумя нулями с каждой стороны даст карту активаций 28 × 28. Узнать размер будущей карты активаций можно с помощью следующего уравнения:

$$\text{Карта активаций} = \frac{D - F + 2P}{S} + 1, \tag{10.3}$$

[1] Термин *рецептивное поле* заимствован непосредственно из области изучения биологических систем зрения, таких как глаза.

где:

- D — размер изображения (ширина или высота, в зависимости от того, что вычисляется — высота или ширина карты активаций);
- F — размер фильтра;
- P — величина дополнения;
- S — длина шага.

То есть для дополнения $P = 2$ мы получим размеры карты активаций 28 × 28:

$$\text{Карта активаций} = \frac{D - F + 2P}{S} + 1.$$

$$\text{Карта активаций} = \frac{28 - 5 + 2 \times 2}{1} + 1.$$

$$\text{Карта активаций} = 28.$$

Учитывая взаимосвязанность размера ядра, длины шага и величины дополнения, необходимо гарантировать взаимное соответствие этих гиперпараметров при проектировании архитектур CNN. То есть значения гиперпараметров должны гарантировать получение действительного размера для карты активаций, в частности, целочисленного значения. Возьмем, например, сверточный слой с размером ядра 5 × 5, длиной шага 2 и без заполнения. Подставив эти значения в уравнение 10.3, получим размер карты активаций 12.5 × 12.5:

$$\text{Карта активаций} = \frac{D - F + 2P}{S} + 1.$$

$$\text{Карта активаций} = \frac{28 - 5 + 0 \times 2}{1} + 1.$$

$$\text{Карта активаций} = 12.5.$$

Не существует такого понятия, как значение половины активации, поэтому сверточный слой с этими размерами просто невозможно вычислить.

СЛОИ СУБДИСКРЕТИЗАЦИИ

Сверточные слои часто работают в тандеме со слоями другого типа, являющимися основными элементами нейронных сетей компьютерного зрения: *слои субдискретизации* (pooling layers). Они служат для уменьшения общего числа параметров в сети и снижения сложности, тем самым ускоряя вычисления и помогая избежать переобучения.

Как отмечалось в предыдущем разделе, сверточный слой может иметь любое количество ядер. Каждое ядро создает карту активации (размеры которой определяются уравнением 10.3), соответственно, результатом работы сверточного слоя является трехмерный массив карт активаций, причем глубина этого массива соответствует количеству фильтров в сверточном слое. Слой субдискретизации пространственно уменьшает размеры этих карт активаций, но оставляет глубину без изменений.

Подобно сверточным слоям, любой слой субдискретизации имеет такие гиперпараметры, как размер фильтра и длина шага. Так же как сверточный слой, слой субдискретизации скользит по своим входным данным. В каждой конкретной позиции он применяет операцию сокращения данных. Чаще всего для этого используется операция `max`, и такие слои называют слоями субдискретизации по максимальному значению: они сохраняют наибольшее значение (*максимальную* активацию) в рецептивном поле, отбрасывая другие значения (см. рис. 10.7)[1]. Как правило, слой субдискретизации имеет размер фильтра 2 × 2 и длину шага 2[2]. В этом случае в каждой позиции слой оценивает четыре активации и сохраняет максимальное значение, уменьшая размер выборки активаций в 4 раза. Поскольку операция объединения происходит независимо для каждого среза в трехмерном массиве, карта активаций 28 × 28 с глубиной 16 уменьшится до размеров 14 × 14, но сохранит полный набор из 16 срезов.

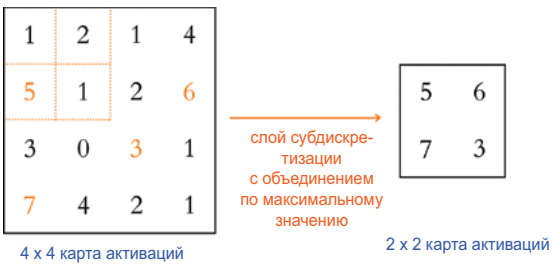


Рис. 10.7. Пример слоя субдискретизации с объединением по максимальному значению, который получает карту активаций 4 × 4. Подобно сверточному слою, слой субдискретизации скользит слева направо и сверху вниз по входной матрице значений. При использовании фильтра размером 2 × 2 слой сохраняет только самое большое из четырех входных значений (например, оранжевое значение «5» в выделенном фрагменте 2 × 2 в верхнем левом углу). При длине шага 2 на выходе из этого слоя получается карта активаций вчетверо меньшего размера: 2 × 2

[1] Существуют также другие варианты объединения (например, *по среднему*, *по норме L2*), но они реже используются на практике, чем объединение по максимальному значению, которое обычно достаточно хорошо подходит для приложений компьютерного зрения и не требует значительных вычислительных ресурсов (например, объединение по среднему требует большего объема вычислений, чем по максимальному значению).

[2] Объединение по максимальному значению с размером фильтра 2 × 2 и длиной шага 2 — это наш выбор по умолчанию. Оба значения, однако, являются гиперпараметрами, с которыми вы можете поэкспериментировать, если хотите.

Альтернативный подход к объединению (субдискретизации) для уменьшения сложности вычислений — использование сверточного слоя с бóльшим шагом (связь длины шага и выходного размера показана в уравнении 10.3). Этот аспект можно с успехом использовать в некоторых специализированных задачах компьютерного зрения (например, в генеративно-состязательных сетях, которые мы будем создавать в главе 12), дающих лучшие результаты без применения слоев субдискретизации. Наконец, вам может быть интересно, что происходит в слое субдискретизации во время обратного распространения: сеть запоминает индекс максимального значения в каждом цикле прямого распространения, который затем используется для вычисления градиента конкретного веса и корректировки правильных параметров.

LENET-5 В KERAS

Еще в главе 1 мы представили иерархическую природу глубокого обучения на рис. 1.11 и обсудили модель компьютерного зрения под названием LeNet-5. В этом разделе мы используем библиотеку Keras и построим модель классификации цифр из коллекции MNIST, основанную на этой знаковой архитектуре. При этом мы добавим в разработку Яна Лекуна и его коллег 1998 года некоторые современные навороты:

- Поскольку современная вычислительная техника обладает большей вычислительной мощностью, мы используем сверточные слои с большим числом ядер. В частности, мы включили 32 и 64 фильтра в первый и второй сверточные слои соответственно, тогда как в исходной модели LeNet-5 было только 6 и 16 фильтров.
- Также благодаря большим вычислительным возможностям мы выполним субдискретизацию активаций только один раз (использовав слой с объединением по максимальному значению), тогда как в LeNet-5 эта операция выполнялась дважды[1].
- Мы используем такие инновации, как активация ReLU и прореживание, которые еще не были изобретены к времени создания LeNet-5.

Желающие самостоятельно опробовать примеры в интерактивном режиме могут воспользоваться блокнотом Jupyter *lenet_in_keras.ipynb*. Как показано в листинге 10.1, если сравнивать с примером в блокноте *deep_net_in_keras.ipynb* (см. главу 9), у нас появились три новые зависимости.

Две из этих зависимостей — `Conv2D` и `MaxPooling2D` — предназначены для реализации сверточных слоев и слоев субдискретизации с объединением по максимальному значению соответственно. А слой `Flatten` позволяет преоб-

[1] В глубоком обучении наблюдается общая тенденция к уменьшению частоты использования слоев субдискретизации, что, скорее всего, обусловлено ростом вычислительных возможностей.

разовывать многомерные массивы в одномерные. Вскоре мы объясним, зачем это нужно, когда будем конструировать архитектуру модели.

Листинг 10.1. Зависимости для реализации LeNet с использованием Keras

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Conv2D, MaxPooling2D # новая!
from keras.layers import Flatten # новая!
```

Далее так же, как в предыдущих блокнотах с моделями классификации рукописных цифр, производится загрузка данных из набора MNIST (см. листинг 5.2). Однако раньше мы преобразовывали изображения из исходного двумерного представления в одномерное, чтобы передать их в полносвязанную сеть (см. листинг 5.3). Первый скрытый слой в нашей сети, напоминающей LeNet-5, будет сверточным, поэтому мы можем оставить изображения в исходном формате 28×28 пикселов, как показано в листинге 10.2[1].

Листинг 10.2. Сохранение двумерной формы изображений

```
X_train = X_train.reshape(60000, 28, 28, 1).astype('float32')
X_valid = X_valid.reshape(10000, 28, 28, 1).astype('float32')
```

Мы по-прежнему используем метод `astype()` для преобразования цифр из целых чисел в числа с плавающей точкой для последующего масштабирования в диапазон от 0 до 1 (как в листинге 5.4). Как и раньше, мы преобразуем целочисленные метки y в векторы прямого кодирования (листинг 5.5).

Покончив с загрузкой и подготовкой данных, приступаем к определению архитектуры LeNet-подобной модели, как показано в листинге 10.3.

Листинг 10.3. Модель сверточной нейронной сети в духе LeNet-5

```
model = Sequential()

# первый сверточный слой:
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                 input_shape=(28, 28, 1)))

# второй сверточный слой с субдискретизацией и прореживанием:
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

[1] В любых массивах, передаваемых в слой `Conv2D()`, предполагается наличие четвертого измерения. Учитывая монохроматическую природу цифр MNIST, мы используем `1` для аргумента четвертого измерения в вызове `reshape()`. Если бы наши данные были полноцветными изображениями, у нас было бы три канала цвета, и тогда мы передали бы аргумент `3`.

```
model.add(Dropout(0.25))
model.add(Flatten())

# полносвязанный скрытый слой с прореживанием:
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))

# выходной слой:
model.add(Dense(n_classes, activation='softmax'))
```

Все предыдущие классификаторы цифр из коллекции MNIST, приводившиеся в этой книге, были полносвязанными сетями, состоящими только из полносвязанных слоев нейронов. Здесь первые два скрытых слоя — это сверточные слои (`Conv2D`)[1]. Для них мы выбрали следующие настройки:

- Целые числа 32 и 64 соответствуют количеству фильтров в первом и втором сверточных слоях.
- Размер ядра `kernel_size` выбран равным 3 × 3 пиксела.
- В качестве функции активации используется функция `relu`.
- Мы выбрали длину шага по умолчанию — 1 пиксел (по вертикали и горизонтали). Альтернативную длину шага можно указать, передав в `Conv2D` аргумент `strides`.
- Мы используем дополнение по умолчанию: `'valid'`. Это означает, что мы решили воздержаться от дополнения: согласно уравнению 10.3, наша карта активаций (при длине шага 1) будет на 2 пиксела короче и на 2 пиксела у́же входного изображения (например, входное изображение 28 × 28 пикселов превратится в карту активаций 26 × 26). При желании можно передать аргумент `padding = 'same'`, и тогда модель будет дополнять входные изображения нулями, чтобы на выходе получить карту активаций того же размера, что и входные изображения (для входных изображений 28 × 28 будут генерироваться карты активаций 28 × 28).

К входному скрытому слою нейронов мы добавили несколько дополнительных слоев для выполнения вычислительных операций[2]:

---

[1] Мы выбрали `Conv2D()`, потому что свертыванию подвергаются двумерные массивы — изображения. В главе 11 мы используем `Conv1D()` для свертывания одномерных данных (текстовых строк). Также существуют слои `Conv3D()`, но мы не будем обсуждать их в этой книге: они предназначены для свертывания трехмерных массивов, что часто требуется при анализе трехмерных медицинских изображений.

[2] Такие слои, как `MaxPooling2D`, `Dropout` и `Flatten`, не содержат искусственных нейронов, поэтому они не считаются самостоятельными скрытыми слоями, как полносвязанные или сверточные слои. Тем не менее они выполняют важные операции с данными, проходящими через нейронную сеть, и мы можем включать их модель с помощью метода `add()`, подобно слоям с нейронами.

- `MaxPooling2D()` используется для уменьшения сложности вычислений. Так же как в примере на рис. 10.7, с параметром `pool_size`, равным 2 × 2, и `strides` со значением по умолчанию (`None` устанавливает длину шага, равную размеру окна объединения), мы уменьшаем размер карты активаций на три четверти.
- Как рассказывалось в главе 9, слой `Dropout()` снижает риск переобучения.
- Наконец, `Flatten()` преобразует трехмерную карту активаций, сгенерированную слоем `Conv2D()`, в одномерный массив. Это дает нам возможность передать активации на вход полносвязанного слоя, который способен принимать только одномерные массивы.

Как уже отмечалось выше, сверточные слои учатся выявлять пространственные особенности в изображениях. Первый сверточный слой учится выявлять простые элементы, такие как прямые линии с разными ориентациями, а второй сверточный слой объединяет эти простые элементы в более абстрактные представления. Задача полносвязанного слоя, который находится на третьем месте в сети, — рекомбинировать пространственные признаки, выявленные вторым сверточным слоем, некоторым произвольным способом, помогающим различать классы изображений (внутри полносвязанного слоя пространственная структура не сохраняется). Иначе говоря, два сверточных слоя учатся выявлять и маркировать пространственные объекты на изображениях, которые затем передаются в полносвязанный слой, отображающий эти пространственные объекты в определенные классы изображений (например, цифра «3» или цифра «8»). То есть сверточные слои можно рассматривать как экстракторы признаков. Полносвязанный слой получает на входе уже готовые извлеченные объекты, а не исходные пикселы.

После полносвязанного слоя мы добавили прореживание `Dropout()` (чтобы избежать переобучения) и завершили сеть выходным слоем softmax — идентичным выходным слоям, которые использовались во всех предыдущих блокнотах с моделями классификации рукописных цифр из коллекции MNIST. Наконец, вызов `model.summary()` выводит сводную информацию о сверточной сети, как показано на рис. 10.8.

Разберем первой колонку Output Shape (Выходная форма) на рис. 10.8:

- Первый сверточный слой `conv2d_1` принимает изображения цифр из коллекции MNIST, имеющие размеры 28 × 28 пикселов. При выбранных гиперпараметрах ядра (размер фильтра, длина шага и размер дополнения) слой выводит карту активаций 26 × 26 (согласно уравнению 10.3)[1]. С 32 ядрами общая карта активаций будет иметь глубину 32.

[1] $\text{Карта активаций} = \frac{D - F + 2P}{S} + 1 = \frac{28 - 3 + 2 \times 0}{1} + 1 = 26.$

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1179776
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

Рис. 10.8. Сводная информация о нашей сверточной сети, реализованной по образу и подобию LeNet-5. Обратите внимание, что для каждого слоя указан размер пакета None (размер пакета для стохастического градиентного спуска). Поскольку размер пакета указывается позже (в вызове метода model.fit()), на этом промежуточном этапе указывается значение None

- Второй сверточный слой принимает карту активаций 26 × 26 × 32 от первого сверточного слоя. Для этого ядра используются те же гиперпараметры, поэтому карта активаций снова сожмется, теперь до 24 × 24. Однако глубина ее увеличится вдвое, потому что в слое используется 64 ядра.
- Как отмечалось выше, слой субдискретизации с объединением по максимальному значению размером ядра 2 и длиной шага 2 уменьшает объем данных, проходящих через сеть, наполовину в каждом из пространственных измерений, что дает в результате карту активаций 12 × 12. Глубина карты активаций слоем субдискретизации не изменяется, поэтому она сохранит свои 64 среза.
- Слой `Flatten` преобразует трехмерную карту активаций в одномерный массив с 9216 элементами[1].
- Полносвязанный скрытый слой содержит 128 нейронов, поэтому на выходе он возвращает одномерный массив со 128 значениями активаций.
- Аналогично выходной слой softmax состоит из 10 нейронов и выводит 10 вероятностей — по одной оценке $\hat{y}$ для каждой возможной цифры.

Теперь обсудим колонку Param # (Число параметров) на рис. 10.8.

[1] $12 \times 12 \times 64 = 9216$.

- Первый сверточный слой имеет 320 параметров:
  - 288 весов: 32 фильтра × 9 весов в каждом (размер фильтра 3 × 3 и 1 канал);
  - 32 смещения, по одному для каждого фильтра.
- Второй сверточный слой имеет 18 496 параметров:
  - 18 432 веса: 64 фильтра × 9 весов в каждом, каждый принимает значения от 32 из предыдущего слоя;
  - 64 смещения, по одному для каждого фильтра.
- Полносвязанный скрытый слой имеет 1 179 776 параметров:
  - 1 179 648 весов: 9216 входов из предыдущего слоя `Flatten` × 128 нейронов в полносвязанном слое[1];
  - 128 смещений, по одному на каждый нейрон.
- Выходной слой имеет 1290 параметров:
  - 1280 весов: 128 входов из предыдущего слоя × 10 нейронов в выходном слое;
  - 10 смещений, по одному на каждый нейрон.
- В совокупности вся сверточная сеть имеет 1 199 882 параметра, подавляющее большинство которых (98.3%) находятся в полносвязанном слое.

Чтобы скомпилировать модель, мы, как обычно, вызываем метод `model.compile()`. Аналогично для обучения вызываем метод `model.fit()`[2]. Результаты лучшей эпохи показаны на рис. 10.9. Ранее лучший результат был достигнут в блокноте *deep_net_in_keras.ipynb* — точность на проверочных данных составила 97.87%. А сеть, созданная в духе LeNet-5, достигла точности 99.27%. Это довольно примечательно, потому что сверточная сеть удалила 65.7% оставшихся ошибок[3]; по всей видимости, эти образцы цифр, теперь опознанных правильно, являются одними из самых трудных для классификации, потому что не были правильно определены глубокой сетью в блокноте *deep_net_in_keras.ipynb*, показавшей весьма внушительный результат.

```
Epoch 9/10
60000/60000 [==============================] - 39s 654us/step - loss: 0.0276 - acc: 0.9911 - val_loss: 0.0260 - val_acc: 0.9927
```

Рис. 10.9. Наша сверточная сеть, реализованная по образу и подобию LeNet-5, достигла пика точности 99.27% на проверочных данных после девяти эпох обучения и превзошла точность полносвязанных сетей, представленных выше в этой книге

---

[1] Обратите внимание, что полносвязанный слой имеет на два порядка больше параметров, чем сверточные слои!

[2] Эти шаги идентичны шагам в предыдущих блокнотах, за небольшим исключением: количество эпох уменьшено до 10, потому что мы в ходе экспериментов обнаружили, что потери на этапе проверки перестали уменьшаться после девяти эпох обучения.

[3] 1 — (100% — 99.27%) / (100% — 97.87%).

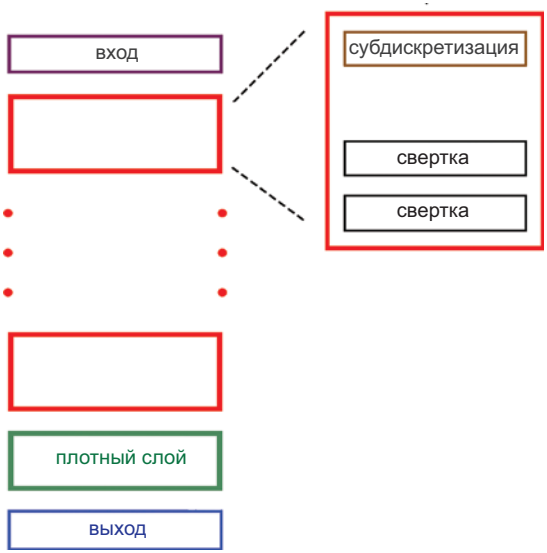


Рис. 10.10. Обобщенное представление архитектуры сверточных нейронных сетей: блок (показан красным) сверточных слоев (часто от одного до трех) и слой субдискретизации повторяются несколько раз. Затем следует один полносвязанный слой (или несколько)

ALEXNET И VGGNET В KERAS

В нашу LeNet-подобную архитектуру (листинг 10.3) мы включили пару сверточных слоев и вслед за ними добавили слой субдискретизации с объединением по максимальному значению. Это обычный подход к организации сверточных нейронных сетей. Как показано на рис. 10.10, сверточные слои (обычно от одного до трех) часто группируют со слоем субдискретизации. Такие блоки могут затем повторяться несколько раз. Как и LeNet-5, такие сверточные архитектуры, как правило, завершаются одним или несколькими полносвязанными скрытыми слоями, а затем выходным слоем.

Модель AlexNet (рис. 1.17), которую мы представили как победительницу конкурса 2012 года в решении задач компьютерного зрения и провозвестницу революции глубокого обучения, является еще одной архитектурой, реализующей блочный подход, представленный на рис. 10.10. В нашем блокноте *alexnet_in_keras.ipynb* мы используем код, показанный в листинге 10.4, который имитирует эту структуру[1].

[1] Это та же модель AlexNet, что была представлена Джейсоном Йосински в его видеоролике DeepViz. Если вы еще не посмотрели этот видеоролик, когда мы упомянули его выше в этой главе, то советуем сделать это прямо сейчас. Вы найдете видеоролик по адресу *bit.ly/DeepViz*.

Листинг 10.4. Сверточная модель в духе AlexNet

```
model = Sequential()

# первый блок со сверточными и объединяющими слоями:
model.add(Conv2D(96, kernel_size=(11, 11),
          strides=(4, 4), activation='relu',
          input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(BatchNormalization())

# второй блок со сверточными и объединяющими слоями:
model.add(Conv2D(256, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(BatchNormalization())

# третий блок со сверточными и объединяющими слоями:
model.add(Conv2D(256, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(384, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(BatchNormalization())

# полносвязанные слои:
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))

# выходной слой:
model.add(Dense(17, activation='softmax'))
```

Вот несколько важных замечаний к этой архитектуре:

- В этом блокноте вместо цифр MNIST мы используем набор полноцветных изображений большего размера (224 × 224 пиксела), чем обусловлена глубина `3` в аргументе `input_shape` в операции создания первого слоя `Conv2D`.
- Модель AlexNet использовала в начальных сверточных слоях фильтры большего размера, чем принято в настоящее время, например `kernel_size = (11, 11)`.
- Такое использование прореживания — только в полносвязанных слоях, близких к выходу модели (а не в более ранних сверточных слоях), — обычное явление. Объясняется это тем, что ранние сверточные слои помогают модели выявить в изображениях пространственные признаки, общие не только для обучающих данных, тогда как более специфические комбинации этих признаков, создаваемые полносвязанными слоями, могут оказаться уникальными для набора обучающих данных и, следовательно, плохо распространяться на проверочные данные.

Модели AlexNet и VGGNet (подробнее о последней рассказывается чуть ниже) очень велики (например, AlexNet имеет 21.9 миллиона параметров), поэтому вам может понадобиться увеличить объем памяти, доступной для Docker на вашей машине, чтобы загрузить их. Инструкции, как это сделать, вы найдете по адресу *bit.ly/DockerMem*.

После того как модель AlexNet была признана победительницей состязания по масштабному распознаванию образов из коллекции ImageNet (ImageNet Large Scale Visual Recognition Challenge, ILSVRC), проводившегося в 2012 году, модели глубокого обучения неожиданно стали широко использоваться в соревнованиях (см. рис. 1.15). Среди этих моделей наблюдалась общая тенденция к увеличению глубины нейронных сетей. Например, в 2014 году второе место в ILSVRC заняла модель VGGNet[1], которая имеет ту же структуру из повторяющихся блоков сверточных и объединяющих слоев, что и AlexNet; VGGNet просто имеет больше таких блоков и использует ядра меньшего размера (все 3 × 3 пиксела). Архитектура этой модели представлена в листинге 10.5, содержащем фрагмент кода из блокнота *vggnet_in_keras.ipynb*.

Листинг 10.5. Сверточная модель в духе VGGNet

```
model = Sequential()

model.add(Conv2D(64, 3, activation='relu',
          input_shape=(224, 224, 3)))
model.add(Conv2D(64, 3, activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(128, 3, activation='relu'))
model.add(Conv2D(128, 3, activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(256, 3, activation='relu'))
model.add(Conv2D(256, 3, activation='relu'))
model.add(Conv2D(256, 3, activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Conv2D(512, 3, activation='relu'))
model.add(Conv2D(512, 3, activation='relu'))
model.add(Conv2D(512, 3, activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())
```

[1] Разработана группой Visual Geometry Group в Оксфордском университете: *Simonyan K., Zisserman A.* (2015). «Very deep convolutional networks for large-scale image recognition». *arXiv: 1409.1556*.

```
model.add(Conv2D(512, 3, activation='relu'))
model.add(Conv2D(512, 3, activation='relu'))
model.add(Conv2D(512, 3, activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(BatchNormalization())

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(17, activation='softmax'))
```

ОСТАТОЧНЫЕ СЕТИ

Как показывают примеры сверточных сетей, приводившиеся выше в этой главе (LeNet-5, AlexNet, VGGNet), наблюдается тенденция к увеличению их глубины. В этом разделе мы вернемся к проблеме затухания градиентов; это (часто сильное) замедление обучения, которое может происходить по мере углубления сетей. Затем опишем остроумное решение, появившееся не так давно: остаточные сети (residual networks).

ЗАТУХАНИЕ ГРАДИЕНТОВ: АХИЛЛЕСОВА ПЯТА ГЛУБОКИХ СВЕРТОЧНЫХ СЕТЕЙ

Модели, обладающие большим количеством слоев, могут выявлять большое разнообразие относительно низкоуровневых признаков в ранних слоях и путем их нелинейного комбинирования получать все более сложные абстракции в более поздних слоях. Этот подход, однако, имеет ограничения: если продолжать увеличивать глубину сетей (например, добавляя все больше и больше блоков сверточных и объединяющих слоев из рис. 10.10), они в конечном итоге столкнутся с проблемой затухания градиентов.

Мы познакомились с проблемой затухания градиентов в главе 9; ее суть заключается в том, что параметры в ранних слоях сети находятся слишком далеко от функции стоимости — источника градиента, который распространяется обратно по сети. Поскольку ошибка распространяется в обратном направлении и с каждым слоем все больше и больше параметров вносят свой вклад в нее, слои, расположенные ближе ко входу, получают все более слабое корректирующее воздействие. В результате с увеличением глубины сетей обучение ранних слоев становится все более затруднительным (см. рис. 8.8).

Из-за проблемы затухания градиентов часто можно наблюдать, что с увеличением глубины сети точность увеличивается до некоторой точки насыщения, а затем начинает падать, когда сеть становится слишком глубокой. Представьте мелкую сеть, которая показывает хорошие результаты. Теперь скопируем эти

слои с их весами и поместим за ними новые слои, чтобы сделать модель глубже в надежде, что новая, более глубокая модель извлечет существенную выгоду из наличия ранее обученных слоев и улучшит точность. Если бы новые слои выполняли простое тождественное отображение (то есть точно воспроизводили бы результаты более ранних слоев), мы не увидели бы увеличения ошибки обучения. Однако, как оказывается, слои в глубоких сетях конкурируют за изучение тождественных функций[1,2]. То есть эти новые слои либо добавляют новую информацию и уменьшают ошибку, либо не добавляют новой информации (но и не выполняют простого тождественного отображения), и ошибка увеличивается. Учитывая, что добавление полезной информации является чрезвычайно редким результатом (относительно базового уровня, который, по сути, является случайным шумом), после определенного момента эти дополнительные слои почти наверняка будут способствовать общему снижению качества модели.

ОСТАТОЧНЫЕ СВЯЗИ

Остаточные сети (residual networks) основаны на идее остаточных связей в так называемых остаточных модулях. Остаточный модуль, как показано на рис. 10.11, это собирательный термин, обозначающий последовательность сверток, операций пакетной нормализации и активаций ReLU, которые завершаются остаточной связью. Для простоты будем рассматривать все эти слои в остаточном модуле как единую дискретную единицу. Согласно наиболее простому определению, остаточная связь образуется, когда вход одного такого остаточного модуля суммируется с его выходом для получения окончательной активации этого модуля. Иначе говоря, остаточный модуль получает некоторые входные данные a_{i-1}[3], которые преобразуются функциями свертки и активации в остаточном модуле в выходные данные a_i, а затем выходные и входные данные суммируются: $y_i = a_i + a_{i-1}$[4].

В описании структуры и математического аппарата остаточной связи из предыдущего абзаца можно заметить одну интересную особенность: если остаточный модуль имеет активацию $a_i = 0$ — то есть он ничего не выявил, — конечным результатом остаточного модуля будут входные данные благодаря суммированию, следуя уравнению, которое мы использовали совсем недавно:

---

[1] *Hardt M. and Ma T.* (2018). «Identity matters in deep learning». *arXiv:1611.04231.*

[2] Держитесь крепче! Чуть ниже мы дополнительно поясним термины «тождественное отображение» и «тождественные функции».

[3] Не забывайте, что входные данные любого конкретного слоя являются выходными данными предыдущего слоя, поэтому здесь они обозначены как a_{i-1}.

[4] Мы обозначили конечный результат всего остаточного модуля как y_i, но это не означает, что он обязательно является конечным результатом всей модели. Мы просто старались избежать путаницы с активациями текущего и предыдущих слоев, указав, что конечный результат является отдельной сущностью, полученной суммированием этих активаций.

$$\begin{aligned} y_i &= a_i + a_{i-1} = \\ &= 0 + a_{i-1} = \\ &= a_{i-1}. \end{aligned}$$

В этом случае остаточный модуль фактически является *тождественной функцией* (identity function). Остаточные модули либо изучают что-то полезное и способствуют снижению ошибок сети, либо выполняют *тождественное отображение* (identity mapping) и вообще ничего не делают. Из-за этой их особенности остаточные связи также называют обходными связями (skip connections), потому что позволяют информации миновать (обходить стороной) функции в остаточном модуле.

В дополнение к этой характеристике *нейтрально, или лучше* остаточных, сетей также следует подчеркнуть ценность их множественности. Рассмотрим схему на рис. 10.12. Когда в сети имеется несколько остаточных модулей, более поздние получают все более сложные комбинации, созданные остаточными модулями и обходными связями в более ранних слоях. Как иллюстрирует представление дерева решений на рисунке справа, в каждом из трех остаточных модулей, имеющихся в сети, информация может либо проходить через остаточный блок, либо миновать его через обходную связь. То есть, как показано внизу на рисунке, для трех остаточных модулей существует восемь возможных путей распространения информации. На практике этот процесс обычно не так разветвлен, как на этом рисунке. Значение a_i редко бывает равно 0, и, следовательно, выход обычно представляет собой некоторую смесь тождественной функции и остаточного модуля. Учитывая это, остаточные сети можно рассматривать как сложные комбинации, или ансамбли, из множества более мелких сетей, которые объединяются на различных глубинах.

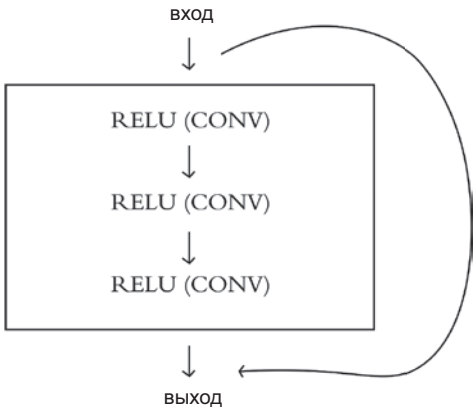


Рис. 10.11. Схематическое представление остаточного модуля. Слои пакетной нормализации и прореживания здесь не показаны, но могут быть включены

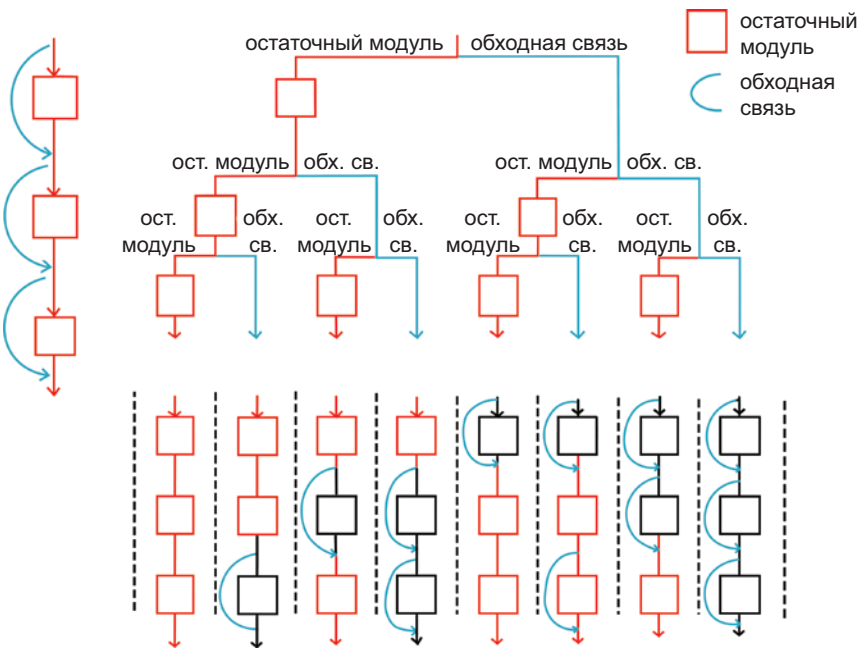


Рис. 10.12. Слева показано традиционное представление остаточных блоков в остаточной сети. Справа — развернутый вид, демонстрирующий, как, в зависимости от использования обходных связей, может изменяться путь передачи информации от входа к выходу

RESNET

Первая глубокая остаточная сеть, ResNet, была представлена Microsoft Research в 2015 году[1] и заняла первое место в состязании ILSVRC того года. Как показано на рис. 1.15, ResNet стала лидером среди алгоритмов глубокого обучения, превзошедших возможности человека в распознавании изображений в 2015 году.

Все, что мы говорили до сих пор, можно было расценить так, будто классификация изображений является *единственным* состязанием в ILSVRC, но на самом деле ILSVRC включает несколько категорий состязаний в области компьютерного зрения, таких как *обнаружение объектов* и *сегментация изображений* (подробнее об этих двух задачах компьютерного зрения мы расскажем далее в этой главе). В 2015 году ResNet заняла первое место не только в классификации изображений ILSVRC, но и в категориях обнаружения объектов и сегментации изображений. В том же году ResNet была признана чемпионом в состязаниях по обнаружению и сегментации, включающих набор изображений под названием COCO, который является альтернативой набору ILSVRC[2].

[1] *He K.* et al. (2015). Deep residual learning for image recognition. *arXiv:1512.03385*.

[2] cocodataset.org

Учитывая богатые трофеи, завоеванные остаточными сетями в состязаниях по компьютерному зрению, совершенно очевидно, что они были прогрессивной инновацией. Им удалось выжать больше сока, чем другим существующим сетям, сделав возможным создание очень глубоких архитектур, не страдающих проблемой снижения точности из-за того, что дополнительные слои не могут извлечь полезную информацию.

В этой книге мы старались сделать примеры кода максимально доступными для читателей, предлагая архитектуры и наборы данных, которые достаточно малы, чтобы можно было проводить обучение даже на небольшом ноутбуке. К сожалению, остаточные сети и наборы данных, на которых они могут продемонстрировать свои преимущества, не попадают в эту категорию. Тем не менее, используя мощный и универсальный подход, называемый *переносом обучения* (transfer learning), который будет представлен в конце этой главы, мы дадим вам ресурсы, чтобы вы могли воспользоваться преимуществами очень глубоких архитектур, таких как ResNet, с моделями, предварительно обученными на больших наборах данных.

ПРИМЕНЕНИЯ КОМПЬЮТЕРНОГО ЗРЕНИЯ

В этой главе вы познакомились с новыми типами слоев, которые прекрасно подходят для реализации моделей компьютерного зрения. Мы также обсудили некоторые способы улучшения этих моделей и рассмотрели ряд канонических алгоритмов компьютерного зрения, изобретенных в последние несколько лет, и задачу классификации изображений, то есть определения основного объекта на изображении, как показано на рис. 10.13 слева. Теперь, чтобы завершить эту главу, мы перейдем к другим интересным применениям компьютерного зрения, помимо классификации изображений. Первое из них — обнаружение объектов, как на втором кадре слева на рис. 10.13, когда алгоритму поручается нарисовать рамки вокруг объектов на изображении. Следующее применение — сегментация изображения, как показано на третьем и четвертом кадрах на рис. 10.13. *Семантическая сегментация* выявляет все объекты определенного класса, вплоть до уровня пикселов, тогда как *сегментация экземпляров* различает разные *экземпляры* конкретного класса, также на уровне пикселов.

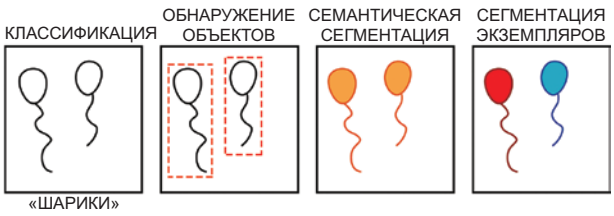


Рис. 10.13. Примеры различных применений компьютерного зрения. Выше в этой главе мы уже встречались с задачей классификации, а теперь рассмотрим обнаружение объектов, семантическую сегментацию и сегментацию экземпляров

ОБНАРУЖЕНИЕ ОБЪЕКТОВ

Представьте фотографию группы людей, сидящих за обеденным столом. На фотографии запечатлено несколько человек. Посередине стола стоит блюдо с жареной курицей и, может быть, бутылка вина. Если бы мы захотели создать автоматизированную систему, определяющую, что было подано на обед, или идентифицирующую людей, сидящих за столом, алгоритм классификации изображений не смог бы обеспечить такой уровень детализации, зато с этой задачей прекрасно справится алгоритм *обнаружения объектов*.

Обнаружение объектов имеет широкую сферу применения, например, для обнаружения пешеходов в поле зрения автопилота автомобиля или для выявления аномалий на медицинских снимках. Вообще говоря, у обнаружения объектов две задачи: собственно обнаружение (*где* на изображении находятся объекты) и классификация (*что* изображено). Обычно этот конвейер имеет три этапа:

1. Выявление области, представляющей интерес.
2. Автоматическое извлечение признаков из этой области.
3. Классификация области.

К числу основных моделей, определивших направления развития в этой сфере, относятся: R-CNN, Fast R-CNN, Faster R-CNN и YOLO.

R-CNN

Модель R-CNN была предложена в 2013 году Россом Гиршиком (Ross Girshick) и его коллегами из Калифорнийского университета в Беркли[1]. Алгоритм моделировал механизм внимания человеческого мозга, который сканирует всю сцену, а затем фокусируется на определенных областях, представляющих интерес. Для подражания этому поведению Гиршик и его коллеги разработали модель R-CNN, которая:

1. Выполняет выборочный поиск *интересных областей* (Regions Of Interest, ROI) на изображении.
2. Извлекает признаки из этих областей с помощью сверточной нейронной сети.
3. Комбинирует два традиционных (как показано на рис. 1.12) подхода машинного обучения — *линейную регрессию* и *метод опорных векторов*, чтобы уточнить расположение ограничительных рамок[2] и классифицировать объекты в каждой из этих рамок соответственно.

---

[1] *Girshnick R.* et al. (2013). «Rich feature hierarchies for accurate object detection and semantic segmentation». *arXiv: 1311.2524*.

[2] Примеры таких ограничительных рамок показаны на рис. 10.14.

Модели R-CNN существенно изменили положение дел в обнаружении объектов, дав огромный прирост точности по сравнению с предыдущей лучшей моделью в состязании Pattern Analysis, Statistical Modeling and Computational Learning (PASCAL) Visual Object Classes (VOC) — анализ шаблонов, статистическое моделирование и вычислительное обучение для определения классов визуальных объектов[1]. Это положило начало эпохе глубокого обучения в обнаружении объектов. Однако эта модель имела некоторые ограничения:

- она была негибкой: на вход можно было передать только одно изображение фиксированного размера;
- она была медленной и требовала значительных вычислительных ресурсов: и обучение, и прогнозирование являются многоступенчатыми процессами с участием сверточной нейронной сети, моделей линейной регрессии и опорных векторов.

Fast R-CNN

Чтобы устранить основной недостаток R-CNN — низкую скорость, Гиршик продолжил разработку и создал модель Fast R-CNN[2]. Он заметил, что на втором шаге алгоритм R-CNN без необходимости несколько раз запускал сверточную нейронную сеть, по одному для каждой интересной области. В Fast R-CNN, как и прежде, на первом шаге выполняется поиск ROI, но на втором шаге сверточная сеть запускается только один раз для обзора его изображения, и извлеченные ею признаки используются сразу для всех интересных областей. Вектор признаков извлекается из конечного слоя CNN, который затем на третьем шаге передается в полносвязанную сеть вместе с ROI. Эта сеть учится концентрироваться только на признаках, которые применяются ко всем интересным областям, и для каждой выводит два результата:

1. Вероятность softmax для категорий классификации (для прогнозирования принадлежности обнаруженного объекта).
2. Ограничитель-регрессор (для уточнения местоположения интересной области).

В соответствии с этим подходом модель Fast R-CNN должна извлекать признаки с использованием CNN только один раз для данного изображения (и тем самым уменьшить сложность вычислений), а затем поиск ROI и полносвязанные слои работают вместе над завершением задачи обнаружения объектов. Как следует из названия, снижение вычислительной сложности модели Fast R-CNN способствует более высокой скорости вычислений. Она также является единой унифицированной моделью, а не множеством независимых компонентов, как

---

[1] Состязания PASCAL VOC проводились с 2005 по 2012 год; набор данных остается доступным и считается одним из золотых стандартов для задач обнаружения объектов.

[2] *Girshnick R.* (2015). «Fast R-CNN». *arXiv: 1504.08083*

ее предшественница. Тем не менее, как и в модели с R-CNN, начальный этап (поиск ROI) в Fast R-CNN все еще является помехой.

Faster R-CNN

Модели в этом разделе — это остроумные и новаторские разработки, в отличие от их названий. Наш третий алгоритм обнаружения объектов — Faster R-CNN, который (как вы уже наверняка догадались!) работает быстрее, чем Fast R-CNN.

Модель Faster R-CNN была изобретена в 2015 году Шаоцином Реном (Shaoqing Ren) и его коллегами из Microsoft Research (на рис. 10.14 показаны примеры результатов ее работы)[1]. Чтобы устранить слабое место в поиске ROI, характерное для R-CNN и Fast R-CNN, Рен и его коллеги нашли остроумное решение и использовали на этом шаге карты активаций из модели CNN. Карты активаций содержат много контекстной информации об изображении. Поскольку каждая карта имеет два измерения, представляющих местоположение, их можно рассматривать буквально как *карты* признаков для данного изображения.

Рис. 10.14. Примеры обнаружения объектов (алгоритмом Faster R-CNN на четырех отдельных изображениях). В каждой интересной области — определяются ограничивающими рамками на изображениях — алгоритм классифицирует объект, находящийся в ней

[1] *Ren S.* et al. (2015). «Faster R-CNN: Towards real-time object detection with region proposal networks». *arXiv: 1506.01497*.

Если, как показано на рис. 10.6, сверточный слой имеет 16 фильтров, карта активаций, которую он выводит, включает 16 карт, представляющих местоположение 16 признаков во входном изображении. Карты признаков как таковые содержат богатую информацию о том, что находится на изображении и *где* это находится. Faster R-CNN использует этот богатый источник информации для определения интересных областей и позволяет сверточной сети беспрепятственно выполнять все три шага процесса обнаружения объектов, тем самым обеспечивая унифицированную архитектуру, основанную на R-CNN и Fast R-CNN, но заметно более быструю.

YOLO

Во всех моделях обнаружения объектов, описанных выше, сверточная нейронная сеть фокусировалась на отдельных интересных областях, а не на всем входном изображении[1]. Джозеф Редмон (Joseph Redmon) и его коллеги в своей статье с описанием алгоритма You Only Look Once (YOLO), опубликованной в 2015 году, сломали эту тенденцию[2]. Сначала алгоритм YOLO извлекает признаки с помощью *предварительно обученной*[3] сверточной сети. Затем изображение делится на ряд ячеек, и для каждой прогнозируется количество ограничивающих рамок и вероятности классификации объектов. Ограничивающие рамки выбираются в соответствии с вероятностями классов выше порогового значения, которые комбинируются для поиска объектов в изображении.

Метод YOLO можно рассматривать как объединение множества ограничивающих рамок меньшего размера, но только если они имеют достаточно высокую вероятность присутствия объекта какого-либо класса. Алгоритм превзошел по скорости Faster R-CNN, но старался точно обнаружить мелкие объекты на изображении. Начиная с оригинальной статьи YOLO, Редмон и его коллеги выпустили свои модели YOLO9000[4] и YOLOv3[5]. Модель YOLO9000 показала высокую скорость и точность, а YOLOv3 уступила в скорости, но показала более высокую точность, в значительной степени благодаря увеличенной сложности базовой архитектуры модели. Обсуждение деталей выходит за рамки этой книги, но отметим, что в этих моделях были использованы самые современные на тот момент алгоритмы обнаружения объектов.

---

[1] Технически сверточная сеть просматривает изображение целиком, в самом начале, как в Fast R-CNN и в Faster R-CNN. Однако в двух предыдущих моделях это был однократный шаг для извлечения признаков, после которого изображение обрабатывалось как набор более мелких областей.

[2] *Redmon J.* et al. (2015). «You Only Look Once: Unified, real-time object detection». *arXiv: 1506.02640*.

[3] Предварительно обученные модели используются в методе переноса обучения, о котором мы расскажем в конце главы.

[4] *Redmon J.* et al. (2016). «YOLO9000: Better, faster, stronger». *arXiv: 1612.08242*.

[5] *Redmon J.* (2018). «YOLOv3: An incremental improvement». *arXiv: 1804.02767*.

СЕГМЕНТАЦИЯ ИЗОБРАЖЕНИЙ

Когда человек рассматривает некоторую сцену с множеством перекрывающихся объектов, например игру в футбол, как показано на рис. 10.15, его мозг в течение нескольких сотен миллисекунд без особых усилий различает фигуры и фон, определяя их границы и отношения между ними. В этом разделе мы рассмотрим *сегментацию изображений* — еще одну область, где глубокое обучение за несколько последних лет преодолело разрыв в способностях между людьми и машинами. Мы остановимся на двух выдающихся архитектурах — Mask R-CNN и U-Net, способных уверенно классифицировать объекты на изображении на уровне пикселов.

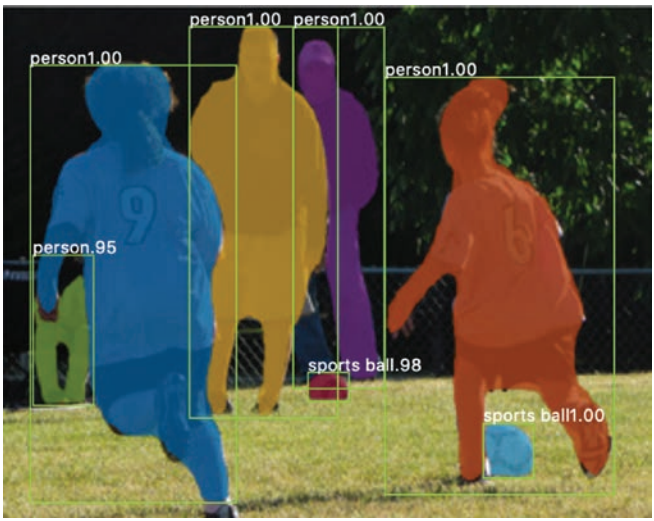


Рис. 10.15. Пример сегментации изображения (результат работы алгоритма Mask R-CNN). Если обнаружение объектов определяет их местоположение с помощью грубых ограничивающих рамок, то сегментация изображений предсказывает местоположение объектов на уровне пикселов

Mask R-CNN

Алгоритм Mask R-CNN был разработан в Facebook AI Research (FAIR) в 2017 году[1]. Он использует:

1. Существующую архитектуру Faster R-CNN для выявления интересных областей на изображении, которые могут содержать объекты.
2. Классификатор ROI, предсказывающий тип объекта в ограничивающей рамке и уточняющий расположение и размер ограничивающей рамки.

[1] *He K.* et al. (2017). «Mask R-CNN». *arXiv: 1703.06870.*

3. Ограничивающие рамки для получения частей карт признаков из сверточной сети, соответствующих данной части изображения.
4. Карту признаков каждой интересной области как входные данные для сверточной сети, которая выводит *маску*, указывающую, какие пикселы соответствуют объекту на изображении. Пример такой маски, состоящей из ярких цветов для обозначения пикселов, принадлежащих отдельными объектам, показан на рис. 10.15.

Задачи сегментации изображений требуют использования бинарных масок в роли меток для обучения. Они состоят из массивов тех же размеров, что и исходное изображение, но вместо RGB-значений пикселов содержат нули и единицы, указывающие, где в изображении находится объект, причем единицы представляют попиксельное положение данного объекта (а нули — все остальное). Если изображение содержит дюжину разных объектов, для него должна иметься дюжина бинарных масок.

U-Net

Другой популярной моделью сегментации изображений является U-Net, которая была разработана в Университете Фрайберга (и упоминалась в конце главы 3, когда мы обсуждали конвейеры автоматической обработки фотографий)[1]. Модель U-Net была создана с целью сегментации биомедицинских изображений и на момент ее создания в 2015 году превосходила лучшие из доступных методов решения двух задач, предложенных на Международном симпозиуме по биомедицинским изображениям[2].

Модель U-Net состоит из полностью сверточной архитектуры, которая начинается со *сжимающего* пути (contracting path), который создает последовательность меньших и более глубоких карт активаций, применяя несколько этапов свертки и объединения по максимальному значению. Следующий за ним *разжимающий* путь (expanding path) восстанавливает эти карты активации до полного разрешения, выполняя несколько этапов повышения дискретизации и свертки. Эти два пути — сжимающий и разжимающий — симметричны (образуют U-образную форму), благодаря чему карты активаций, полученные в сжимающем пути, могут объединяться с полученными в разжимающем пути.

Сжимающий путь позволяет модели выявить мелкие признаки на изображении. Они затем передаются непосредственно на вход разжимающего пути. К концу разжимающего пути, как ожидается, модель локализует эти признаки на изображении окончательных размеров. После передачи карт признаков из

[1] *Ronneberger O.* et al. (2015). «U-Net: Convolutional networks for biomedical image segmentation». *arXiv: 1505.04597*.

[2] К этим двум задачам относятся задачи сегментации нейронных структур на снимках, полученных с помощью электронного микроскопа, и выделения клеток.

сжимающего пути в разжимающий последующий сверточный слой позволяет сети научиться точно выявлять и локализовать эти признаки. В конечном итоге получается сеть, хорошо справляющаяся с идентификацией объектов и определением их местоположения в двумерном пространстве.

ПЕРЕНОС ОБУЧЕНИЯ

Для достижения приемлемой эффективности многие модели, описанные в этой главе, требуется обучать на очень больших наборах различных изображений. Для такого обучения требуются значительные вычислительные ресурсы, да и сбор коллекций изображений — сложная и дорогостоящая процедура. В ходе обучения сверточная сеть учится извлекать из изображений общие признаки. На низком уровне это линии, края, цвета и простые формы; на более высоком — текстуры, комбинации форм, части объектов и другие сложные визуальные элементы (см. рис. 1.17). Если достаточно глубокую сверточную сеть обучить на подходящем наборе разнообразных изображений, ее карты признаков почти наверняка будут содержать богатую библиотеку визуальных элементов, которые можно собрать и объединить, чтобы сформировать практически любое изображение. Например, карту признаков, идентифицирующую текстуру с ямочками, можно объединить с картой, распознающей круглые объекты, и еще одной картой, реагирующей на белый цвет, и получить инструмент идентификации мяча для гольфа. *Перенос обучения* использует преимущества такой библиотеки визуальных элементов, содержащихся в картах признаков предварительно обученной сверточной сети, и повторно использует их, чтобы научиться идентифицировать новые классы объектов.

Представьте, к примеру, что вы решили построить модель компьютерного зрения, решающую задачу бинарной классификации, к которой мы неоднократно обращались, начиная с главы 6: различение хот-догов и объектов, не являющихся хот-догами. Конечно, вы можете разработать большую и сложную сверточную сеть, которая принимает изображения хот-догов и... в общем, не хот-догов, и выводит прогноз класса с использованием сигмоидной функции активации. Вы можете обучить эту модель на большом количестве изображений и ожидать, что ранние сверточные слои выявят набор карт признаков, свойственных хот-догам. Честно говоря, такая сеть будет работать очень хорошо. Но понадобится много времени и вычислительных ресурсов, чтобы правильно обучить сверточную сеть, и потребуется большое количество разнообразных изображений, чтобы она смогла создать достаточно разнообразный набор карт признаков. И тут на сцену выходит перенос обучения: вместо обучения модели с самого начала вы можете воспользоваться возможностями глубокой модели, уже обученной на большом наборе изображений, и быстро переориентировать ее специально для идентификации хот-догов.

Выше в этой главе мы упоминали VGGNet как пример классической модели компьютерного зрения. В листинге 10.5 и в нашем блокноте Jupyter *vggnet_*

in_keras.ipynb мы демонстрировали модель VGGNet16, состоящую из 16 слоев искусственных нейронов — в основном повторяющихся блоков свертки и объединения (см. рис. 10.10). С ней тесно связана модель VGGNet19, которая имеет дополнительный блок свертки и объединения (с тремя сверточными слоями) и послужит нам отправной точкой в знакомстве с переносом обучения. В блокноте *transfer_learning_in_keras.ipynb* мы загружаем модель VGGNet19 и модифицируем ее для своих целей.

Главное преимущество VGG19 перед VGG16 состоит в том, что дополнительные слои в модели VGG19 обеспечивают ей дополнительные возможности для абстрактного представления визуальных образов. Главный же ее недостаток заключается в том, что эти дополнительные слои увеличивают общее число параметров и, следовательно, время обучения. Кроме того, из-за проблемы затухания градиентов возникают дополнительные сложности с обратным распространением через дополнительные ранние слои VGG19.

Для начала выполним стандартные инструкции импортирования и загрузим предварительно обученную модель VGGNet19 (листинг 10.6).

Листинг 10.6. Загрузка модели VGGNet19 для переноса обучения

```
# Загрузка зависимостей:
from keras.applications.vgg19 import VGG19
from keras.models import sequential
from keras.layers import Dense, Dropout, Flatten
from keras.preprocessing.image import ImageDataGenerator

# Загрузка предварительно обученной модели VGG19:
vgg19 = VGG19(include_top=False,
              weights='imagenet',
              input_shape=(224,224,3),
              pooling=None)

# Зафиксировать все слои в базовой модели VGGNet19:
for layer in vgg19.layers:
    layer.trainable = False
```

Для удобства библиотека Keras уже включает готовую сетевую архитектуру с параметрами (которые называются *весами*, но также включают смещения), поэтому загрузка предварительно обученной модели выполняется легко и просто[1]. Аргументы, передаваемые в функцию `VGG19`, помогают определить некоторые характеристики загруженной модели:

- `include_top = False` указывает, что из исходной архитектуры VGGNet19 не требуется загружать заключительные полносвязанные слои класси-

[1] В числе других предварительно обученных моделей в состав библиотеки Keras входит также архитектура ResNet, которая была представлена выше в этой главе. Полный список моделей, готовых к загрузке, вы найдете по адресу: keras.io/applications.

фикации. Эти слои были обучены для классификации исходных данных ImageNet. Вместо них, как вы увидите ниже, мы создадим свои слои и обучим их сами, используя собственные данные.

- `weights = 'imagenet'` требует загрузить параметры модели, полученные при обучении на наборе данных ImageNet, включающем 14 миллионов образцов[1].
- `input_shape = (224,224,3)` задает размеры входных изображений для обучения идентификации хот-догов.

После загрузки модели выполняется цикл `for`, который обходит все слои модели и устанавливает в них флаг `trainable` в значение `False`, чтобы *предотвратить* обновление параметров в этих слоях во время обучения. Мы знаем, что сверточные слои в VGGNet19 были обучены на большом наборе данных ImageNet для представления обобщенных визуальных признаков, поэтому предохраняем базовую модель от изменений.

В листинге 10.7 мы добавляем новые полносвязанные слои поверх базовой модели VGGNet19. Эти слои будут принимать признаки, извлеченные из входного изображения предварительно обученными сверточными слоями, и учиться использовать эти признаки для идентификации изображений хот-догов.

Листинг 10.7. Добавление слоев классификации в модель переноса обучения

```
# Создание последовательной модели и добавление в нее модели VGG19:
model = Sequential()
model.add(vgg19)

# Добавление новых слоев поверх модели VGG19:
model.add(Flatten(name='flattened'))
model.add(Dropout(0.5, name='dropout'))
model.add(Dense(2, activation='softmax', name='predictions'))

# Подготовка (компиляция) модели к последующему обучению:
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Далее мы используем экземпляр класса `ImageDataGenerator` для загрузки данных (листинг 10.8). Этот класс определен в библиотеке Keras и служит для загрузки изображений на лету. Он особенно удобен, когда нет возможности загрузить сразу все обучающие данные в память или когда во время обучения происходит произвольное обогащение данных в режиме реального времени[2].

---

[1] Единственным другим значением, которое можно было передать в параметре `weights` на момент написания этих строк, было значение `'None'`, означающее случайную инициализацию, но в будущем могут быть доступны другие наборы параметров модели, полученные при обучении на других наборах данных.

[2] В главе 9 мы говорили, что обогащение данных — это эффективный способ увеличения размера набора обучающих данных, помогающий модели лучше обобщать ранее не встречавшиеся данные.

Листинг 10.8. Определение генераторов данных

```
# Создание двух экземпляров класса-генератора:
train_datagen = ImageDataGenerator(
    rescale=1.0/255,
    data_format='channels_last',
    rotation_range=30,
    horizontal_flip=True,
    fill_mode='reflect')

valid_datagen = ImageDataGenerator(
    rescale=1.0/255,
    data_format='channels_last')

# Определение размера пакета:
batch_size=32

# Определение генераторов обучающих и проверочных данных:
train_generator = train_datagen.flow_from_directory(
    directory='./hot-dog-not-hot-dog/train',
    target_size=(224, 224),
    classes=['hot_dog','not_hot_dog'],
    class_mode='categorical',
    batch_size=batch_size,
    shuffle=True,
    seed=42)

valid_generator = valid_datagen.flow_from_directory(
    directory='./hot-dog-not-hot-dog/test',
    target_size=(224, 224),
    classes=['hot_dog','not_hot_dog'],
    class_mode='categorical',
    batch_size=batch_size,
    shuffle=True,
    seed=42)
```

Генератор обучающих данных будет поворачивать случайно выбранные изображения на угол в пределах 30 градусов или переворачивать по горизонтали, масштабировать данные в диапазон от 0 до 1 (умножая на 1/255) и загружать результаты в массивы, имеющие формат «последнее измерение описывает каналы»[1]. Генератору проверочных данных нужно будет только изменять масштаб изображений; обогащать данные в нем не имеет смысла. Наконец, метод `flow_from_directory()` сообщает каждому генератору каталог, откуда тот должен загружать изображения[2]. Остальные аргументы этого метода не нуждаются в пояснениях.

---

[1] Заглянув в листинг 10.6, можно увидеть, что модель принимает входные данные с размерами 224 × 224 × 3, то есть размерность, определяющая число каналов, следует последней. Также размерность с числом каналов можно было бы указать первой.

[2] Инструкции по загрузке данных включены в наш блокнот Jupyter.

Теперь все готово к обучению (листинг 10.9). На этот раз вместо метода `fit()`, как во всех предыдущих примерах обучения моделей в этой книге, мы используем метод `fit_generator()`, потому что вместо массивов данных будем передавать генератор[1]. В процессе обучения этой модели мы определили, что лучшей оказалась шестая эпоха, в которой модель достигла точности 81.2%.

Листинг 10.9. Обучение модели

```
model.fit_generator(train_generator, steps_per_epoch=15,
                    epochs=16, validation_data=valid_generator,
                    validation_steps=15)
```

Это демонстрирует силу метода переноса обучения. Затратив совсем немного времени на обучение и почти не затрачивая его на выбор архитектуры и настройку гиперпараметров, мы получили модель, которая достаточно хорошо справляется с довольно сложной задачей классификации изображений: идентификацией хот-догов. Уделив время на настройку гиперпараметров, результаты можно улучшить.

КАПСУЛЬНЫЕ СЕТИ

В 2017 году Сара Сабур (Sara Sabour) и ее коллеги из команды Джеффа Хинтона (рис. 1.16) в Google Brain в Торонто представили новую идею под названием *капсульные сети* (capsule networks)[2]. Капсульные сети вызвали большой интерес, потому что они оказались способны учитывать информацию о местоположении. Сверточные нейронные сети, к большому сожалению, не обладают такой способностью, поэтому, например, сверточная

Рис. 10.16. Для сверточных нейронных сетей, которые не учитывают взаимного расположения визуальных элементов, обе фигуры, слева и справа, могут быть классифицированы как лицо Джеффа Хинтона. Капсульные сети, напротив, учитывают информацию о местоположении и поэтому с меньшей вероятностью примут правую фигуру за лицо

[1] Как мы предупреждали выше в этой главе в разделе с описанием моделей AlexNet и VGGNet, работая с очень большими моделями, можно столкнуться с нехваткой памяти. Посетите страницу bit.ly/DockerMem, где рассказывается, как увеличить объем памяти, доступной контейнеру Docker. Кроме того, можете попробовать уменьшить гиперпараметр `batch_size`.

[2] *Sabour S.* et al. (2017). «Dynamic routing between capsules». *arXiv: 1710.09829*.

сеть будет классифицировать оба изображения на рис. 10.16 как человеческое лицо. Обсуждение теории капсульных сетей выходит за рамки этой книги, но специалисты по компьютерному зрению, как правило, знают о них, поэтому мы просто хотели, чтобы вы тоже имели в виду их существование. В настоящее время они слишком требовательны к вычислительным ресурсам, чтобы занять доминирующие позиции в практическом применении, но с удешевлением вычислений и появлением новых теоретических достижений эта ситуация скоро изменится.

ИТОГИ

В этой главе вы познакомились со сверточными слоями, специализирующимися на выявлении пространственных структур, что делает их особенно полезными для задач компьютерного зрения. Мы использовали эти слои для создания сверточной сети по образу и подобию классической архитектуры LeNet-5, что позволило превзойти точность распознавания рукописных цифр полносвязанными сетями, которые мы создавали во второй части. В завершение главы мы обсудили лучшие приемы построения сверточных сетей и рассмотрели наиболее значимые применения алгоритмов машинного зрения. В следующей главе вы узнаете, что возможности пространственного распознавания образов в сверточных слоях хорошо подходят не только для компьютерного зрения, но и для других задач.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым:

параметры:

- вес w;
- смещение b;

- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
  - Linear;
- входной слой;
- скрытый слой;
- выходной слой;

- типы слоев:
  - плотный (полносвязанный);
  - softmax;
  - сверточный;
  - субдискретизации с объединением по максимальному значению;
  - преобразования в плоский массив;
- функции стоимости (потерь):
  - квадратичная (среднеквадратичная ошибка);
  - перекрестная энтропия;
- прямое распространение;
- обратное распространение;
- нестабильность градиентов (в частности, затухание);
- метод Глоро инициализации весов;
- пакетная нормализация;
- прореживание;
- оптимизаторы:
  - стохастический градиентный спуск;
  - Adam;
- гиперпараметры оптимизаторов:
  - скорость обучения η;
  - размер пакета.

11

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

В главе 2 мы познакомились с представлением естественного языка, пригодным для использования в вычислениях, и в частности с *векторами слов* — мощным подходом, помогающим количественно выразить значения слова. В настоящей главе мы рассмотрим приемы, которые позволят вам создавать свои векторы слов и использовать их в качестве входных данных в моделях глубокого обучения.

Модели обработки естественного языка, которые мы создадим в этой главе, будут включать уже известные нам слои нейронов: полносвязанные слои из глав 5–9 и сверточные слои из главы 10. Наши модели NLP также будут включать новые типы слоев, в том числе использующиеся в рекуррентных нейронных сетях (Recurrent Neural Networks, RNN). Рекуррентные сети изначально создавались для обработки информации, встречающейся в таких последовательностях, как тексты на естественном языке, но в действительности они более универсальны и способны обрабатывать *любые* последовательные данные (например, изменение финансовых показателей или температуры в данном географическом местоположении с течением времени). В конце главы вы найдете раздел о сетях глубокого обучения, обрабатывающих данные посредством нескольких параллельных потоков. Эта идея значительно расширяет возможности творческого подхода к разработке архитектуры модели и, как вы увидите, также может повысить точность модели.

ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ДАННЫХ НА ЕСТЕСТВЕННОМ ЯЗЫКЕ

Для повышения точности моделирования к данным на естественном языке можно применить несколько видов предварительной обработки. На практике обычно выполняются следующие шаги:

- *Лексемизация* (tokenization): весь документ (например, книга) разбивается на список отдельных элементов языка (например, слов), которые называют *лексемами*.
- Преобразование всех символов в *нижний регистр*: слова в начале предложений, начинающиеся с заглавной буквы (например, *Она*) имеют то же значение, что и в середине предложения (*она*). Преобразуя все символы в корпусе в нижний регистр, мы игнорируем деление на заглавные и строчные буквы.
- Удаление *стоп-слов*: стоп-словами называют часто встречающиеся слова, которые, как правило, не имеют большого значения, например «и», «что», «который» и «поэтому». В отношении точного перечня стоп-слов нет единого мнения, и в зависимости от сферы применения бывает целесообразно считать (или не считать) определенные элементы стоп-словами. Например, в этой главе мы создадим модель классификации отзывов о фильмах на положительные и отрицательные. В некоторые списки стоп-слов включены отрицания, такие как «нет», «не», и «не может», что может затруднить определение смысла отзыва о фильме, поэтому такие слова, вероятно, не следует считать стоп-словами.
- Удаление *знаков препинания*: обычно учет знаков препинания не добавляет точности модели естественного языка, поэтому их часто просто удаляют.
- Стемминг (stemming)[1]: стемминг — это усечение слов до их *основы* (stem). Например, слова «дом» и «домашний» имеют одну основу — «дом». Стемминг, особенно для небольших наборов данных, может оказаться выгодным, потому что позволяет объединить слова с одинаковыми значениями в одну лексему. В этом случае появляется больше примеров контекста лексемы, что позволяет таким методам, как word2vec или GloVe, точнее определять подходящее местоположение лексемы в векторном пространстве слов (см. рис. 2.5 и 2.6).
- Обработка *n-грамм*: некоторые слова используются вместе настолько часто, что их комбинацию лучше рассматривать как единое понятие, чем несколько отдельных понятий. Например, *New York* — это *биграмма* (*n*-грамма с длиной, равной двум), *New York City* — *триграмма* (*n*-грамма с длиной, равной трем). Объединение слов *new*, *york* и *city* имеет определенное значение, которое, вероятно, лучше зафиксировать в виде одной лексемы (и одной точкой в векторном пространстве слов) вместо трех.

В зависимости от конкретной задачи, для которой разрабатывается модель, а также от набора исходных данных, можно использовать все, некоторые или

[1] *Лемматизация*, более сложная альтернатива стеммингу, требует использования справочного словаря. Для наших целей стемминга будет вполне достаточно, чтобы представлять несколько родственных слов в виде одной лексемы.

ни один из этих видов предварительной обработки. Рассматривая возможность применения любой предварительной обработки в конкретной задаче, можете использовать свою интуицию, чтобы оценить, будет ли эта обработка полезной. Вот некоторые рекомендации, часть из которых уже приводилась выше:

- Стемминг может пригодиться при использовании небольших корпусов, но почти не имеет ценности для больших.
- Аналогично преобразование всех символов в нижний регистр тоже может пригодиться при использовании небольших корпусов, но в большом корпусе, где намного выше вероятность использования слов в разных контекстах, различие между словами, например «генеральный» (прилагательное, означающее «общий», как в словосочетании «генеральный план») и «генерал» (существительное, означающее воинское звание), может оказаться ценным.
- Удаление знаков препинания вообще не дает преимуществ ни в одном из случаев. Представьте, например, алгоритм выбора ответа на вопрос, который может использовать вопросительные знаки как помощь в идентификации вопросов.
- Интерпретация отрицаний как стоп-слов может принести пользу в некоторых классификаторах, но, например, не в классификаторе эмоциональной окраски. Выбор элементов для включения в список стоп-слов может иметь решающее значение для вашего конкретного приложения, поэтому будьте внимательны с этим. Во многих случаях лучше удалять только ограниченное подмножество стоп-слов.

Если вы не уверены, будет ли полезен тот или иной вид предварительной обработки, исследуйте проблему опытным путем, включив ее и посмотрев, влияет ли она на точность модели глубокого обучения. Как правило, чем больше корпус, тем меньше предварительной обработки для него требуется. При использовании небольшого корпуса вас, возможно, будет беспокоить вероятность встретить редкие или не вошедшие в словарь вашего обучающего набора данных слова. Объединяя несколько редких слов в одну лексему, вы почти наверняка сможете повысить эффективность обучения модели значениям групп связанных слов. Однако по мере увеличения корпуса редкие и не вошедшие в словарь слова будут представлять все менее и менее серьезную проблему. Поэтому, если корпус очень большой, может быть полезно *отказаться* от объединения нескольких слов в одну лексему, так как в корпусе будет достаточное число экземпляров, чтобы эффективно моделировать их уникальное значение, а также относительно тонкие нюансы взаимоотношений между словами (которые иначе могли бы быть объединены).

Практические примеры применения предварительной обработки вы найдете в нашем блокноте Jupyter *natural_language_preprocessing.ipynb*. Он начинается с загрузки ряда зависимостей:

```
import nltk
from nltk import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from nltk.stem.porter import *
nltk.download('gutenberg')
nltk.download('punkt')
nltk.download('stopwords')

import string

import gensim
from gensim.models.phrases import Phraser, Phrases
from gensim.models.word2vec import Word2Vec

from sklearn.manifold import TSNE

import pandas as pd
from bokeh.io import output_notebook, output_file
from bokeh.plotting import show, figure
%matplotlib inline
```

Большая часть этих зависимостей связана с библиотеками *nltk* (Natural Language Toolkit) и *genism* (еще одна библиотека средств обработки естественного языка для Python). Мы объясним использование каждой отдельной зависимости в описании следующих примеров кода.

ЛЕКСЕМИЗАЦИЯ

В этом блокноте мы использовали обучающий набор данных на основе нескольких книг, не защищенных авторскими правами, из *Проекта Гутенберг* (Project Gutenberg)[1]. Этот корпус распространяется в составе библиотеки nltk, поэтому его легко загрузить, как показано в следующем фрагменте кода:

```
from nltk.corpus import gutenberg
```

Этот маленький корпус включает 18 литературных произведений, в том числе роман «Эмма» Джейн Остин, сказку «Алиса в Стране чудес» Льюиса Кэрролла и три пьесы малоизвестного парня по имени Уильям Шекспир. (Выполните команду `gutenberg.fileids()`, чтобы вывести названия всех 18 документов.) Выполнив команду `len(gutenberg.words())`, вы увидите, что корпус состоит из 2.6 миллиона слов — при таком количестве вполне можно опробовать все примеры кода в этом разделе на ноутбуке.

---

[1] Проект Гутенберг получил свое название в честь изобретателя печатного станка Йоханнеса Гутенберга (Johannes Gutenberg) и содержит десятки тысяч электронных книг. Эти книги являются классическими литературными произведениями со всего мира, авторские права на которые уже истекли, что делает их свободно доступными. См. gutenberg.org.

Корпус можно преобразовать в список предложений, вызвав метод `sent_tokenize()` из библиотеки nltk:

```
gberg_sent_tokens = sent_tokenize(gutenberg.raw())
```

Обратившись к первому элементу списка командой `gberg_sent_tokens[0]`, вы увидите, что первым в корпусе Проекта Гутенберг является роман «Эмма», потому что первый элемент содержит титульную страницу книги, номер главы и первое предложение, следующие друг за другом через символы новой строки (`\n`):

```
'[Emma by Jane Austen 1816]\n\nVOLUME I\n\nCHAPTER I\n\n\nEmma Woodhouse,
handsome, clever, and rich, with a comfortable home\nand happy
disposition, seemed to unite some of the best blessings\nof existence;
and had lived nearly twenty-one years in the world\nwith very little to
distress or vex her.'
```

Во втором элементе находится отдельное предложение, которое можно получить командой `gberg_sent_tokens[1]`:

```
"She was the youngest of the two daughters of a most affectionate,
\nindulgent father; and had, in consequence of her sister's marriage,\
nbeen mistress of his house from a very early period."
```

Можно пойти еще дальше и с помощью метода `word_tokenize()` из библиотеки nltk преобразовать это предложение в список слов:

```
word_tokenize(gberg_sent_tokens[1])
```

Этот вызов выведет список слов, попутно отбросив все пробельные символы, включая символы новой строки (см. рис. 11.1). Слово *father* (отец), например, занимает 15-ю позицию в списке слов из второго предложения, в чем можно убедиться, выполнив следующий код:

```
word_tokenize(gberg_sent_tokens[1])[14]
```

Методы `sent_tokenize()` и `word_tokenize()` могут пригодиться вам для работы с собственными данными на естественном языке, но для работы с корпусом Проекта Гутенберг удобнее использовать встроенный метод `sents()`, позволяющий получить тот же результат за один шаг:

```
gberg_sents = gutenberg.sents()
```

Эта команда создает список списков `gberg_sents`. Список верхнего уровня состоит из отдельных предложений, каждое из которых содержит список слов более низкого уровня. Соответственно, метод `sents()` также разделит титульный лист и названия глав на отдельные элементы, в чем можно убедиться, вызвав `gberg_sents[0:2]`:

```
[['[', 'Emma', 'by', 'Jane', 'Austen', '1816', ']'],
['VOLUME', 'I'],
['CHAPTER', 'I']]
```

```
['She',
 'was',
 'the',
 'youngest',
 'of',
 'the',
 'two',
 'daughters',
 'of',
 'a',
 'most',
 'affectionate',
 ',',
 'indulgent',
 'father',
 ';',
 'and',
 'had',
 ',',
 'in',
 'consequence',
 'of',
 'her',
 'sister',
 "'",
 's',
 'marriage',
 ',',
 'been',
 'mistress',
 'of',
 'his',
 'house',
 'from',
 'a',
 'very',
 'early',
 'period',
 '.']
```

Рис. 11.1. Второе предложение из классического романа «Эмма» Джейн Остин, преобразованное в список слов

Поэтому первое фактическое предложение из романа «Эмма» находится в четвертом элементе в `gberg_sents`, и для доступа к 15-му слову (*father*) во втором предложении следует использовать ссылку `gberg_sents[4][14]`.

ПРЕОБРАЗОВАНИЕ ВСЕХ СИМВОЛОВ В НИЖНИЙ РЕГИСТР

Остальные виды предварительной обработки данных на естественном языке мы будем применять итеративно к каждому предложению в отдельности. К концу этого раздела мы применим каждый вид обработки ко всему корпусу из 18 документов.

Взглянув на рис. 11.1, можно заметить, что первое слово в предложении — *She* — начинается с заглавной буквы. Чтобы не учитывались различия между заглавными и строчными буквами и слова *she* и *She* интерпретировались как идентичные, можно воспользоваться методом `lower()` из библиотеки `string`, как показано в листинге 11.1.

Листинг 11.1. Преобразование символов предложения в нижний регистр

```
[w.lower() for w in gberg_sents[4]]
```

Эта строка кода вернет тот же список, что изображен на рис. 11.1, только первым элементом списка теперь будет слово *she*, а не *She*.

УДАЛЕНИЕ СТОП-СЛОВ И ЗНАКОВ ПРЕПИНАНИЯ

Другой потенциальный недостаток предложения на рис. 11.1 — присутствие большого числа стоп-слов и знаков препинания. Чтобы исправить это, используем оператор + и объединим предопределенный список английских стоп-слов в библиотеке nltk со списком знаков препинания в библиотеке `string`:

```
stpwrds = stopwords.words('english') + list(string.punctuation)
```

Изучив содержимое созданного списка `stpwrds`, можно увидеть, что он содержит много общих слов, которые обычно не имеют особого значения, таких как *a*, *an* и *the*[1]. Однако в нем также присутствуют такие слова, как *not* и другие, выражающие отрицание, которые важны, например, для классификации эмоциональной окраски предложения «This film was not good» («Этот фильм получился не очень хорошим»).

В любом случае, чтобы удалить из предложения все элементы, присутствующие в `stpwrds`, можно использовать *генератор списков*[2] (листинг 11.2), который выполняет преобразование символов в нижний регистр, как мы делали это в листинге 11.1.

Листинг 11.2. Удаление стоп-слов и знаков препинания с помощью генератора списков

```
[w.lower() for w in gberg_sents[4] if w.lower() not in stpwrds]
```

---

[1] Эти три конкретных слова в английском языке называются *артиклями*, или *определителями*.

[2] Введение в генераторы списков в языке Python можно найти по адресу bit.ly/listComp.

Если сравнивать с рис. 11.1, эта строка кода вернет гораздо более короткий список, содержащий только слова, передающие значительную долю смысла:

```
['youngest',
 'two',
 'daughters',
 'affectionate',
 'indulgent',
 'father',
 'consequence',
 'sister',
 'marriage',
 'mistress',
 'house',
 'early',
 'period']
```

СТЕММИНГ

Для стемминга слов можно использовать алгоритм Porter[1], реализованный в библиотеке nltk. Для этого создадим экземпляр `PorterStemmer()` и добавим вызов его метода `stem()` в генератор списков из листинга 11.2, как показано в листинге 11.3.

Листинг 11.3. Добавление стемминга слов в генератор списков

```
[stemmer.stem(w.lower()) for w in gberg_sents[4]
 if w.lower() not in stpwrds]
```

Этот код выведет следующий результат:

```
['youngest',
 'two',
 'daughter',
 'affection',
 'indulg',
 'father',
 'consequ',
 'sister',
 'marriag',
 'mistress',
 'hous',
 'earli',
 'period']
```

Он похож на предыдущий результат, только на этот раз многие слова сокращены до основы:

[1] *Porter M. F.* (1980). «An algorithm for suffix stripping». Program, 14, 130–7.

1. Слово *daughters* (дочери) сократилось до *daughter* (дочь) — слова во множественном и единственном числе интерпретируются идентично.
2. Слово *house* сократилось до его основы *hous* — однокоренные слова, такие как *house* и *housing*, интерпретируются идентично.
3. Слово *early* сократилось до его основы *earli* — слова с разной сравнительной степенью, такие как *early* (рано), *earlier* (раньше), *earliest* (самый ранний), интерпретируются идентично.

Стемминг может принести дополнительную пользу при обработке таких небольших корпусов, как наш, содержащих относительно небольшое число примеров каждого конкретного слова. Объединяя похожие слова, мы получаем больше вхождений каждой объединенной версии, благодаря чему ей может быть присвоено более точное местоположение в векторном пространстве (см. рис. 2.6). Однако в больших корпусах, когда имеется много примеров даже редких слов, может быть выгодно по-разному интерпретировать слова во множественном и единственном числе, однокоренные слова и сравнительные степени; небольшие отличия могут оказаться весьма ценными для передачи смысла.

ОБРАБОТКА N-ГРАММ

Чтобы интерпретировать биграммы, такие как *New York*, как одну лексему вместо двух, можно использовать методы `Phrases()` и `Phraser()` из библиотеки gensim. Как показано в листинге 11.4, они используются следующим образом:

1. `Phrases()` применяется для обучения «детектора» и помогает ему определить, как часто та или иная пара слов встречается в корпусе (такие часто встречающиеся биграммы называют *bigram collocation* — устойчивыми биграммами) относительно частоты вхождения каждого слова из пары по отдельности.
2. `Phraser()` берет устойчивые биграммы, обнаруженные объектом `Phrases()`, и использует их для создания объекта, который эффективно преобразует все устойчивые биграммы в одну лексему.

Листинг 11.4. Выявление устойчивых биграмм

```
phrases = Phrases(gberg_sents)
bigram = Phraser(phrases)
```

Если выполнить этот код, он сохранит в свойство `bigram.phrasegrams` словарь со счетчиком и *оценкой устойчивости* каждой биграммы. На рис. 11.2 показаны первые несколько записей из этого словаря.

Каждой биграмме на рис. 11.2 соответствуют свои значения счетчика и оценки устойчивости. Например, биграмма *two daughters* (две дочери) встречается в корпусе Гутенберга всего 19 раз. Она имеет довольно низкую оценку устойчивости

(12.0), означающую, что слова *two* и *daughters* встречаются не очень часто по сравнению с частотой их вхождения по отдельности. Напротив, биграмма *Miss Taylor* (мисс Тейлор) встречается намного чаще (48 раз), при этом слова *Miss* и *Taylor* встречаются вместе гораздо чаще, чем по отдельности (оценка 453.8).

```
{(b'two', b'daughters'): (19, 11.966813731181546),
 (b'her', b'sister'): (195, 17.7960829227865),
 (b"'", b's'): (9781, 31.066242737744524),
 (b'very', b'early'): (24, 11.01214147275924),
 (b'Her', b'mother'): (14, 13.529425062715127),
 (b'long', b'ago'): (38, 63.22343628984788),
 (b'more', b'than'): (541, 29.023584433996874),
 (b'had', b'been'): (1256, 22.306024648925288),
 (b'an', b'excellent'): (54, 39.063874851750626),
 (b'Miss', b'Taylor'): (48, 453.75918026073305),
 (b'very', b'fond'): (28, 24.134280468850747),
 (b'passed', b'away'): (25, 12.35053642325912),
 (b'too', b'much'): (173, 31.376002029426687),
 (b'did', b'not'): (935, 11.728416217142811),
 (b'any', b'means'): (27, 14.096964108090186),
 (b'wedding', b'-'): (15, 17.4695197740113),
 (b'Her', b'father'): (18, 13.129571562488772),
 (b'after', b'dinner'): (21, 21.5285481168817),
```

Рис. 11.2. Словарь с биграммами, найденными в корпусе

Просматривая содержимое словаря на рис. 11.2, обратите внимание, что в нем присутствуют биграммы, включающие слова, начинающиеся с заглавных букв, и знаки препинания. Мы решим эти проблемы в следующем разделе, а пока посмотрим, как созданный объект `bigram` можно использовать для преобразования каждой биграммы в одну лексему. Для примера разберем короткое предложение — строку символов с пробелами, используя метод `split()`:

```
tokenized_sentence = "Jon lives in New York City".split()
```

Если теперь вывести `tokenized_sentence`, мы увидим простой список унограмм: `['Jon', 'lives', 'in', 'New', 'York', 'City']`. Но если передать этот список объекту `bigram` как `bigram[tokenized_sentence]`, в списке появится лексема для биграммы *New York*: `['Jon', 'lives', 'in', 'New_York', 'City']`.

После выявления биграмм в корпусе с помощью объекта `bigram` можно повторно пропустить новый корпус, заполненный биграммами, через методы `Phrases()` и `Phraser()` и выявить триграммы (такие, как *New York City*). Эту операцию можно повторить и выявить тетраграммы (и снова чтобы выявить пентаграммы и т. д.); однако ценность каждого следующего шага уменьшается. Для большинства применений вполне достаточно определить биграммы (и иногда триграммы). Кстати, попытавшись выявить триграммы в корпусе из Проекта Гутенберг, вы вряд ли обнаружите триграмму *New York City*. В корпусе классической литературы это название упоминается недостаточно часто.

ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ПОЛНОГО КОРПУСА

Посмотрев несколько примеров предварительной обработки отдельных предложений, реализуем ее на всем корпусе Проекта Гутенберг. При этом мы добавим биграммы в корпус, не содержащий заглавных букв или знаков препинания.

Ниже в этой главе мы используем корпус отзывов к фильмам, составленный Эндрю Маасом (Andrew Maas) с коллегами из Стэнфордского университета для определения эмоциональной окраски отзывов с помощью моделей NLP[1]. На этапах предварительной обработки данных Маас и его коллеги решили не отбрасывать стоп-слова, потому что они «служат индикаторами эмоциональной окраски»[2]. Они также решили не использовать стемминг, посчитав свой корпус достаточно большим, чтобы их модель NLP смогла «выявить схожие представления слов с одинаковой основой». Проще говоря, в результате обучения модели слова с одинаковым значением должны оказаться рядом в векторном пространстве (см. рис. 2.6).

Следуя их указаниям, мы тоже воздержимся от удаления стоп-слов и стемминга при предварительной обработке корпуса из Проекта Гутенберг, как показано в листинге 11.5.

Листинг 11.5. Преобразование букв в нижний регистр и удаление знаков препинания в корпусе из Проекта Гутенберг

```
lower_sents = []
for s in gberg_sents:
    lower_sents.append([w.lower() for w in s if w.lower()
                        not in list(string.punctuation)])
```

В этом примере мы создаем пустой список `lower_sents`, а затем, используя цикл `for`, добавляем в него обработанные предложения[3]. Для предварительной обработки каждого предложения внутри цикла используется генератор списков из листинга 11.2, в данном случае мы просто удаляем знаки препинания и преобразуем все символы в нижний регистр.

После удаления знаков препинания и преобразования букв в нижний регистр можно выполнить поиск устойчивых биграмм в корпусе:

```
lower_bigram = Phraser(Phrases(lower_sents))
```

На этот раз, в отличие от листинга 11.4, мы создали объект `lower_bigram` одной строкой кода, объединив вызовы методов `Phrases()` и `Phraser()`. На рис. 11.3 показаны первые несколько записей из получившегося словаря `lower_bigram.`

---

[1] *Maas A.* et al. (2011). «Learning word vectors for sentiment analysis». Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics, 142–50.

[2] Это соответствует нашим рассуждениям выше в этой главе.

[3] Для предварительной обработки больших корпусов вместо простого (и понятного) цикла `for` мы советуем использовать оптимизированные приемы функционального программирования, поддерживающие распараллеливание операций.

```
{(b'two', b'daughters'): (19, 11.080802900992637),
 (b'her', b'sister'): (201, 16.93971298099339),
 (b'very', b'early'): (25, 10.516998773665177),
 (b'her', b'mother'): (253, 10.70812618607742),
 (b'long', b'ago'): (38, 59.226442015336005),
 (b'more', b'than'): (562, 28.529926612065935),
 (b'had', b'been'): (1260, 21.583193129694834),
 (b'an', b'excellent'): (58, 37.41859680854167),
 (b'sixteen', b'years'): (15, 131.42913000977515),
 (b'miss', b'taylor'): (48, 420.4340982546865),
 (b'mr', b'woodhouse'): (132, 104.19907841850323),
 (b'very', b'fond'): (30, 24.185726346489627),
 (b'passed', b'away'): (25, 11.751473221742694),
 (b'too', b'much'): (177, 30.36309017383541),
 (b'did', b'not'): (977, 10.846196223896685),
 (b'any', b'means'): (28, 14.294148100212627),
 (b'after', b'dinner'): (22, 18.60737125272944),
 (b'mr', b'weston'): (162, 91.63290824201266),
```

Рис. 11.3. Словарь с биграммами, найденными в корпусе после преобразования символов в нижний регистр и удаления знаков препинания

`phrasegrams`: сравнивая эти биграммы с биграммами на рис. 11.2, можно заметить, что они содержат только символы нижнего регистра (например, *miss taylor*) и среди них отсутствуют биграммы со знаками препинания.

Однако при детальном исследовании результатов на рис. 11.3 можно заметить, что минимальные пороговые значения по умолчанию для количества вхождений и оценки устойчивости слишком широки: пары слов вроде *two daughters* (две дочери) и *her sister* (ее сестра) не должны рассматриваться как биграммы. Чтобы выявить более устойчивые биграммы, мы попробовали более низкие пороги для количества вхождений и оценки устойчивости, последовательно удваивая их. Следуя этим путем, мы получили удовлетворительные результаты, передав методу `Phrases()` необязательные аргументы `min_count` со значением 32 и `threshold` со значением 64 (минимальное количество вхождений и порог оценки устойчивости соответственно), как показано в листинге 11.6.

Листинг 11.6. Выявление устойчивых биграмм с более низкими пороговыми значениями

```
lower_bigram = Phraser(Phrases(lower_sents,
                               min_count=32, threshold=64))
```

Конечно, результаты получились не идеальные[1], потому что все еще присутствует несколько сомнительных биграмм, таких как *great deal* (не в пример) и *few minutes* (несколько минут), тем не менее содержимое словаря `lower_bigram.phrasegrams`, как показано на рис. 11.4, теперь в большей степени оправданно.

Получив с помощью листинга 11.6 более удачный объект `lower_bigram`, мы наконец можем использовать цикл `for` и добавить в свой корпус обработанные предложения, как показано в листинге 11.7.

[1] Разумеется, это всего лишь статистические приближения!

Листинг 11.7. Создание корпуса, включающего биграммы

```
clean_sents = []
for s in lower_sents:
    clean_sents.append(lower_bigram[s])
```

```
{(b'miss', b'taylor'): (48, 156.44059469941823),
 (b'mr', b'woodhouse'): (132, 82.04651843976633),
 (b'mr', b'weston'): (162, 75.87438262077481),
 (b'mrs', b'weston'): (249, 160.68485093258923),
 (b'great', b'deal'): (182, 93.36368125424357),
 (b'mr', b'knightley'): (277, 161.74131790625913),
 (b'miss', b'woodhouse'): (173, 229.03802722366902),
 (b'years', b'ago'): (56, 74.31594785893046),
 (b'mr', b'elton'): (214, 121.3990121932397),
 (b'dare', b'say'): (115, 89.94000515807346),
 (b'frank', b'churchill'): (151, 1316.4456593286038),
 (b'miss', b'bates'): (113, 276.39588291692513),
 (b'drawing', b'room'): (49, 84.91494947493561),
 (b'mrs', b'goddard'): (58, 143.57843432545658),
 (b'miss', b'smith'): (58, 73.03442128232508),
 (b'few', b'minutes'): (86, 204.16834974753786),
 (b'john', b'knightley'): (58, 83.03755747111268),
 (b'don', b't'): (830, 250.30957446808512),
```

Рис. 11.4. Словарь с биграммами, полученный с более низкими пороговыми значениями

```
['sixteen',
 'years',
 'had',
 'miss_taylor',
 'been',
 'in',
 'mr_woodhouse',
 's',
 'family',
 'less',
 'as',
 'a',
 'governess',
 'than',
 'a',
 'friend',
 'very',
 'fond',
 'of',
 'both',
 'daughters',
 'but',
 'particularly',
 'of',
 'emma']
```

Рис. 11.5. Предложение из корпуса Гутенберга после предварительной обработки

Для примера на рис. 11.5 показан седьмой элемент обработанного корпуса (`clean_sents[6]`) — предложение, включающее биграммы *miss taylor* и *mr woodhouse*.

СОЗДАНИЕ ВЕКТОРНЫХ ПРЕДСТАВЛЕНИЙ С ПОМОЩЬЮ АЛГОРИТМА WORD2VEC

Получив «очищенный» корпус `clean_sents` на естественном языке, можно приступать к созданию векторного пространства слов из корпуса (рис. 2.6). Как будет показано в этом разделе, их можно создавать с помощью всего одной строки кода. Однако эта строка кода не должна выполняться вслепую, и в ней есть несколько необязательных аргументов, которые следует упомянуть. Поэтому прежде, чем углубляться в пример кода, мы рассмотрим теоретические предпосылки, лежащие в основе векторов слов.

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ АЛГОРИТМА WORD2VEC

В главе 2 мы в общих чертах познакомились с понятием векторов слов и обсудили основную идею «определения смысла слова по его окружению», согласно которой значение любого слова можно представить как среднее значение слов, окружающих его. word2vec — это алгоритм обучения без учителя[1], то есть он не использует никаких меток, которые могут или не могут существовать в корпусе. Это означает, что на вход алгоритма word2vec можно передать любой набор данных на естественном языке[2].

Запуская алгоритм word2vec, есть возможность выбирать между двумя базовыми архитектурами — *skip-gram* (SG) или *continuous bag of words* (CBOW; произносится как *си-боу*) — каждая из которых обычно дает примерно сопоставимые результаты, несмотря на максимизацию вероятностей с «противоположных» точек зрения. Чтобы понять суть, обратимся к примеру нашего «игрушечного» корпуса на рис. 2.5:

```
you shall know a word by the company it keeps
```

В нем мы рассматриваем слово `word` как *целевое*, а три слова справа и слева от него — как *контекстные*. (Это соответствует размеру окна в три слова — один из основных гиперпараметров, который следует учитывать при применении алгоритма word2vec.) В архитектуре SG контекстные слова предсказываются

---

[1] Краткое описание различий между обучением с учителем, без учителя и с подкреплением вы найдете в главе 4.

[2] *Mikolov T.* et al. (2013). «Efficient estimation of word representations in vector space». *arXiv:1301.3781.*

на основе целевого слова[1]. В архитектуре CBOW, наоборот, целевое слово предсказывается на основе контекстных слов[2].

Для более конкретного знакомства с алгоритмом word2vec сосредоточимся на архитектуре CBOW (точно так же мы могли бы опираться на архитектуру SG). Модель с архитектурой CBOW предсказывает целевое слово как среднее из окружающих его контекстных слов, рассматриваемых *совместно*. «Совместно» означает «все сразу»: без учета конкретных позиций контекстных слов относительно друг друга и самого целевого слова. Эта особенность архитектуры отражена в ее названии, «bag of words» — «мешок слов»:

- она берет все контекстные слова в окне, справа и слева от целевого слова;
- бросает (образно говоря!) все эти контекстные слова в мешок. Если это поможет вам запомнить, что последовательность слов не имеет значения, можете даже представить, что она встряхивает мешок;
- вычисляет среднее значение для всех контекстных слов в мешке и на основании среднего оценивает, каким может быть целевое слово.

Если бы нас волновал синтаксис — грамматика языка (см. рис. 2.9, чтобы вспомнить элементы естественного языка), — тогда порядок слов имел бы значение. Но поскольку алгоритм word2vec учитывает только семантику — *значения* слов, — порядок следования контекстных слов не имеет значения для вычисления среднего.

Определив значение части «BOW» аббревиатуры CBOW, перейдем к ее составляющей «continuous» («непрерывный»): скользящее окно, включающее целевое и контекстные слова, *непрерывно* скользит по тексту, слово за словом, от первого слова в корпусе до последнего. В каждой позиции на этом пути целевое слово оценивается с учетом контекстных слов. С помощью стохастического градиентного спуска расположение слов в векторном пространстве может сдвигаться и соответственно, оценки целевых слов могут постепенно улучшаться.

На практике, как показано в табл. 11.1, архитектуру SG предпочтительнее использовать при работе с небольшим корпусом. Она хорошо справляется с представлением редких слов в векторном пространстве. Архитектура CBOW, в свою очередь, намного эффективнее в вычислительном отношении, поэтому ее лучше использовать при работе с очень большим корпусом. Кроме того, CBOW немного лучше представляет часто встречающиеся слова[3].

---

[1] Выражаясь техническим языком машинного обучения, функция стоимости в архитектуре skip-gram максимизирует логарифм вероятности любого возможного контекстного слова из корпуса, опираясь на текущее целевое слово.

[2] И снова, выражаясь техническим языком машинного обучения, функция стоимости в архитектуре CBOW максимизирует логарифм вероятности любого возможного целевого слова из корпуса, опираясь на контекстные слова.

[3] Кроме архитектуры SG или CBOW, алгоритм word2vec позволяет также выбрать метод обучения. Доступно два разных варианта: *иерархический softmax* (hierarchical

Таблица 11.1. Сравнение архитектур word2vec

Архитектура	Прогнозирует	Преимущества
Skip-gram (SG)	Контекстные слова по заданному целевому слову	Лучше подходит для небольших корпусов; хорошо справляется с представлением редких слов
CBOW	Целевое слово по заданным контекстным словам	Намного быстрее; немного лучше справляется с представлением часто встречающихся слов

Алгоритм word2vec является наиболее широко используемым для создания векторных представлений из корпусов на естественном языке, но он не единственный. Самой заметной альтернативой является алгоритм GloVe — глобальные векторы для представления слов, предложенный выдающимися исследователями естественного языка Джеффри Пеннингтоном (Jeffrey Pennington), Ричардом Сошером (Richard Socher) и Кристофером Мэннингом (Christopher Manning)[1]. В тот момент, в 2014 году, они были коллегами и вместе работали в Стэнфордском университете.

GloVe и word2vec различаются методологией, лежащей в их основе: word2vec использует прогностические модели, а GloVe основывается на подсчете. Но как бы то ни было, оба создают векторные пространства, предлагающие аналогичные свойства приложениям NLP. Однако, судя по некоторым исследованиям, в отдельных случаях word2vec обеспечивает несколько лучшие результаты. Одно из потенциальных преимуществ GloVe заключается в том, что этот алгоритм предусматривает возможность распараллеливания и обработки корпуса на нескольких процессорах или даже на нескольких компьютерах, поэтому он будет хорошим выбором, когда требуется создать векторное пространство на основе очень большого корпуса с множеством уникальных слов.

Еще одной заметной альтернативой алгоритмам word2vec и GloVe является fastText[2,3,4,5]. Он был разработан исследователями в Facebook. Основное преимущество fastText в том, что он работает на уровне *подслов* — его векторы «слов» на самом деле являются подкомпонентами слов. Это позволяет алгоритму fastText преодолевать некоторые проблемы, связанные с редкими и не входящими в словарь корпуса словами, которые рассматривались в разделе, посвященном предварительной обработке, в начале этой главы.

---

softmax) и *отрицательная выборка* (negative sampling). Первый предполагает нормализацию и лучше подходит для редких слов. Последний, напротив, не использует нормализацию, что делает его более подходящим для часто встречающихся слов и низкоразмерных векторных пространств. Для нас различия между этими двумя методами обучения незначительны, и мы не будем их подробно рассматривать.

1 *Pennington J.* et al. (2014). «GloVe: Global vectors for word representations». Proceedings of the Conference on Empirical Methods in Natural Language Processing.

2 Библиотека fastText с открытым исходным кодом доступна по адресу fasttext.cc.

3 *Joulin A.* et al. (2016). «Bag of tricks for efficient text classification». *arXiv: 1607.01759*.

4 Bojanowski, P., et al. (2016). «Enriching word vectors with subword information». *arXiv: 1607.04606*.

5 Обратите внимание, что ведущий автор знаковой статьи о word2vec Томас Миколов (Tomas Mikolov) является также соавтором обеих этих статей об алгоритме fastText.

ВЫЧИСЛЕНИЕ ВЕКТОРОВ СЛОВ

Независимо от выбора алгоритма для создания векторов слов — с помощью word2vec или другой альтернативы — есть два важных аспекта, которые стоит принять во внимание при оценке качества векторов слов: *внутренние* и *внешние* оценки.

К внешним относится оценка качества векторов слов в приложениях NLP, например, в классификаторе эмоциональной окраски или, может быть, в инструменте распознавания именованных сущностей. Определение внешних оценок может занять много времени, потому что для этого требуется выполнить все последующие этапы обработки, включая, возможно, обучение дорогостоящей в вычислительном отношении модели глубокого обучения, после которых вы будете уверены, что стоит сохранить изменения в векторах слов, если они заметно улучшают точность вашего приложения.

К внутренним оценкам, напротив, относится оценка качества векторов слов не по окончательному приложению NLP, а по какой-то промежуточной задаче. Примером таких задач является оценка соответствия векторов слов арифметическим аналогиям, подобным тем, что показаны на рис. 2.7. Например, если вы начинаете со «слова-вектора» `король`, вычитаете из него слово `мужчина`, добавляете слово `женщина`, то оказываетесь ли вы рядом со «словом-вектором» `королева`?[1]

В противоположность внешним оценкам, внутренние определяются быстрее. Они также могут помочь лучше понять промежуточные шаги (и устранить неполадки в них) в рамках более широкого процесса NLP. Однако увеличение внутренней оценки может не привести к увеличению точности приложения NLP, если только не была определена надежная, поддающаяся количественной оценке взаимосвязь между качеством промежуточного тестирования и приложением NLP.

ЗАПУСК WORD2VEC

Как упоминалось выше и было показано в листинге 11.8, алгоритм word2vec можно запустить единственной строкой кода, хотя и с несколькими аргументами.

Листинг 11.8. Запуск word2vec

```
model = Word2Vec(sentences=clean_sents, size=64,
                 sg=1, window=10, iter=5,
                 min_count=10, workers=4)
```

[1] Томас Миколов с коллегами разработали тестовый набор из 19 500 таких аналогий в своей статье об алгоритме word2vec, вышедшей в 2013 году. Этот набор доступен по адресу *download.tensorflow.org/data/questions-words.txt*.

Вот описание всех аргументов, которые мы передали в метод `Word2Vec()` из библиотеки gensim:

- `sentences`: служит для передачи корпуса в виде списка списков, такого как `clean_sents`. Элементами верхнего уровня в списке являются предложения, а элементы нижнего уровня могут быть лексемами.
- `size`: количество измерений в векторном пространстве слов, которое должно быть получено алгоритмом word2vec. Это гиперпараметр, который можно изменять и оценивать внешне и внутренне. Как и другие гиперпараметры, упоминавшиеся в этой книге, этот тоже имеет оптимальное значение. Чтобы найти его, можно, например, выбрать для начала 32 измерения и затем последовательно удваивать это число. Удвоение количества измерений удваивает вычислительную сложность модели, но если это ведет к заметно более высокой точности, то дополнительная вычислительная сложность может быть оправданна. Но если можно вдвое уменьшить количество измерений и снизить вычислительную сложность без заметного снижения точности модели NLP, такое уменьшение также можно считать оправданным. Проведя несколько внутренних проверок (о которых мы расскажем ниже), мы выяснили, что в данном случае 64 измерения позволяют получить более разумные векторы слов, чем 32. Однако дальнейшее удвоение числа измерений до 128 не дало заметного улучшения.
- `sg`: если передать в этом аргументе число `1`, будет выбрана архитектура skip-gram, а если оставить значение по умолчанию `0`, то CBOW. Как показано в табл. 11.1, архитектура SG лучше подходит для небольших наборов данных, таких как наш корпус Гутенберга.
- `window`: для архитектуры SG хорошо подойдет размер окна 10 (включающий 20 контекстных слов), поэтому мы выбрали значение `10`. Для архитектуры CBOW оптимальным считается размер окна 5 (10 контекстных слов). В любом случае с этим гиперпараметром можно экспериментировать и оценивать его как внешне, так и внутренне. Небольшие его корректировки, однако, могут не оказать заметного влияния.
- `iter`: по умолчанию метод `Word2Vec()` из библиотеки gensim перебирает содержимое корпуса (то есть скользит по всем его словам) пять раз. Итерации word2vec аналогичны эпохам обучения модели глубокого обучения. В небольшом корпусе, подобном нашему, векторы слов достигают лучшего качества за несколько итераций. С другой стороны, при обработке очень большого корпуса вычислительные затраты даже на две итерации могут оказаться слишком велики, и из-за значительного числа примеров слов в таком корпусе векторы могут не улучшаться с увеличением числа итераций.
- `min_count`: это минимальное число вхождений слова в корпусе для его попадания в векторное пространство. Если данное целевое слово встречается

только один или пару раз, число примеров контекстных слов, которые следует учитывать, окажется слишком ограниченным, и их местоположение в векторном пространстве может оцениваться ненадежно. Поэтому часто разумно ограничивать минимальное количество 10 примерами. Чем выше число, тем меньше словарный запас, который будет доступен задаче NLP. Это еще один гиперпараметр, доступный для настройки, причем внешние оценки, вероятно, будут полезнее, чем внутренние, потому что размер доступного словаря может оказать значительное влияние на приложение NLP.

- `workers`: это количество ядер процессора, которые можно задействовать для обучения. Если процессор вашего компьютера имеет, скажем, восемь ядер, тогда восемь — это наибольшее число параллельных потоков, которые вы сможете запустить. Если в этом случае вы решите использовать менее восьми ядер, то оставите вычислительные ресурсы доступными для других задач.

Мы сохранили нашу модель в репозитории GitHub с примерами к книге, вызвав метод `save()` объекта word2vec:

```
model.save('clean_gutenberg_model.w2v')
```

Чтобы запускать алгоритм word2vec у себя, вы можете загрузить наши векторы слов, используя следующий код:

```
model = gensim.models.Word2Vec.load('clean_gutenberg_model.w2v')
```

Решив использовать наши векторы слов, в следующих примерах вы получите те же результаты, что и мы[1]. Узнать размер словаря можно вызовом `len(model.wv.vocab)`. В данном случае он сообщает, что в нашем корпусе `clean_sents` имеется 10 329 слов (точнее, лексем), встречающихся не менее 10 раз[2]. Одно из слов в этом словаре — *dog* (собака). Как показано на рис. 11.6, мы можем узнать его местоположение в 64-мерном векторном пространстве, обратившись к элементу `model.wv['dog']`.

В качестве элементарной внутренней оценки качества векторов слов можно использовать метод `most_similar()`, который помогает убедиться, что слова с аналогичными значениями находятся по соседству в векторном пространстве[3].

---

[1] Каждый раз, когда запускается алгоритм word2vec, он выбирает случайное начальное местоположение для каждого слова в векторном пространстве. По этой причине для одних и тех же данных и аргументов метод `Word2Vec()` будет генерировать уникальные векторы слов, но семантические отношения должны сохраняться.

[2] Размер словаря равен количеству лексем в корпусе, встречающихся не менее 10 раз, потому что мы установили параметр `min_count = 10` в вызове `Word2Vec()` в листинге 11.8.

[3] Строго говоря, сходство между двумя словами определяется здесь вычислением косинусного сходства.

Например, вот как можно получить три слова, расположенных в векторном пространстве рядом со словом *father* (отец):

```
model.wv.most_similar('father', topn=3)
```

```
array([ 0.38401067,  0.01232518, -0.37594706, -0.00112308,  0.38663676,
        0.01287549,  0.398965  ,  0.0096426 , -0.10419296, -0.02877572,
        0.3207022 ,  0.27838793,  0.62772304,  0.34408906,  0.23356602,
        0.24557391,  0.3398472 ,  0.07168821, -0.18941355, -0.10122284,
       -0.35172758,  0.4038952 , -0.12179806,  0.096336  ,  0.00641343,
        0.02332107,  0.7743452 ,  0.03591069, -0.20103034, -0.1688079 ,
       -0.01331445, -0.29832968,  0.08522387, -0.02750671,  0.32494134,
       -0.14266558, -0.4192913 , -0.09291836, -0.23813559,  0.38258648,
        0.11036541,  0.005807  , -0.16745028,  0.34308755, -0.20224966,
       -0.77683043,  0.05146591, -0.5883941 , -0.0718769 , -0.18120563,
        0.00358319, -0.29351747,  0.153776  ,  0.48048878,  0.22479494,
        0.5465321 ,  0.29695514,  0.00986911, -0.2450937 , -0.19344331,
        0.3541134 ,  0.3426432 , -0.10496043,  0.00543602], dtype=float32)
```

Рис. 11.6. Местоположение лексемы «dog» в 64-мерном векторном пространстве слов, сгенерированном на основе корпуса книг из Проекта Гутенберг

Эта команда выведет:

```
[('mother', 0.8257375359535217),
 ('brother', 0.7275018692016602),
 ('sister', 0.7177823781967163)]
```

Как показывает этот вывод, ближе всего к слову *father* (отец) в этом векторном пространстве располагаются слова *mother* (мать), *brother* (брат) и *sister* (сестра). Иначе говоря, в нашем 64-мерном пространстве ближайшим[1] к слову *father* является *mother*. В табл. 11.2 приводится несколько дополнительных примеров слов, наиболее похожих на конкретные слова, выбранные нами из словаря (то есть наиболее близкие к ним). Все пять примеров выглядят вполне оправданными, учитывая маленький корпус Гутенберга[2].

Предположим, что мы выполнили следующую строку кода:

```
model.wv.doesnt_match("mother father sister brother dog".split())
```

Она выведет `dog`, показывая, что слово *dog* (собака) меньше всего подходит для образования пар со всеми остальными словами. Также можно выполнить следующую строку, чтобы получить оценку сходства слов *father* и *dog* (она равна `0.44`):

```
model.wv.similarity('father', 'dog')
```

[1] То есть имеет самое короткое евклидово расстояние в этом 64-мерном векторном пространстве.

[2] Обратите внимание, что последнее тестовое слово в табл. 11.2 — *ма'ам* (мэм) — стало доступно только после выявления устойчивых биграмм (см. листинги 11.6 и 11.7).

Таблица 11.2. Слова, наиболее похожие на тестовые, выбранные из словаря корпуса Гутенберга

Тестовое слово	Наиболее похожее слово	Оценка косинусного сходства
father (отец)	mother (мать)	0.82
dog (собака)	puppy (щенок)	0.78
eat (есть)	drink (пить)	0.83
day (день)	morning (утро)	0.76
ma_am (мэм)	madam (мадам)	0.85

Эта оценка сходства `0.44` намного ниже, чем между словом *father* и любым из слов — *mother*, *brother* и *sister*, поэтому неудивительно, что слово *dog* оказалось относительно далеко от других четырех слов в нашем векторном пространстве.

В качестве последней внутренней оценки можно попробовать вычислить словесно-векторные аналогии, как на рис. 2.7. Например, чтобы вычислить $v_{father} - v_{man} + v_{woman}$ (*отец – мужчина + женщина*), выполните этот код:

```
model.wv.most_similar(positive=['father', 'woman'], negative=['man'])
```

Ближайшим словом оказывается слово `mother`, являющееся правильным ответом на аналогию. Попробуйте также выполнить этот код (он вычисляет выражение *муж – мужчина + женщина*):

```
model.wv.most_similar(positive=['husband', 'woman'], negative=['man'])
```

В этом случае ближайшим оказывается слово `wife` (жена), то есть снова правильный ответ. Эти результаты позволяют считать, что в целом векторное пространство слов построено правильно.

Заданное измерение в *n*-мерном векторном пространстве слов не обязательно представляет какой-либо конкретный фактор, связывающий слова. Например, реальные различия в значении рода имен существительных или времен глаголов вполне могут быть представлены некоторым направлением (то есть некоторой комбинацией измерений) в векторном пространстве, но это смысловое направление может совпасть — или коррелировать — с определенной осью векторного пространства только чисто случайно.

Это контрастирует с некоторыми другими подходами к созданию *n*-мерных векторных пространств, в которых оси представляют некоторые конкретные объясняющие переменные. Один такой подход, с которым знакомы многие, — метод главных компонент (Principal Component Analysis, PCA), определяющий линейно некоррелированные (ортогональные) векторы, которые вносят вклад в дисперсию в наборе данных. Главное отличие этого метода заключается в том, что первые главные компоненты объясняют наибольшую долю дисперсии в наборе данных, поэтому исследователь может сосредоточиться на них и игнорировать последующие главные компоненты; но в векторном пространстве слов все измерения могут быть важны и должны приниматься во внимание. Такие подходы полезны для уменьшения размерности, потому что не требуют учитывать все измерения.

ОТОБРАЖЕНИЕ ВЕКТОРОВ СЛОВ НА ГРАФИКЕ

Человеческий мозг плохо воспринимает визуальную информацию, имеющую больше трех измерений. Поэтому о построении векторов слов, которые могут иметь десятки или даже сотни измерений, в их исходном формате не может быть и речи. К счастью, существуют методы *уменьшения размерности*, позволяющие приблизительно отобразить местоположения слов из многомерного векторного пространства в двух- или трехмерное. Для такого уменьшения размерности мы рекомендуем использовать подход, который называется *стохастическое представление соседей с использованием t-распределения* (t-distributed stochastic neighbor embedding, t-SNE; произносится как *ти-сни*), разработанный Лоренсом ван дер Маатеном (Laurens van der Maaten) в сотрудничестве с Джеффом Хинтоном (см. рис. 1.16)[1].

В листинге 11.9 приводится код из нашего блокнота *natural_language_preprocessing.ipynb*, сокращающий 64-мерное векторное пространство слов из Проекта Гутенберг до двух измерений, а затем сохраняющий полученные координаты x и y в объекте `DataFrame` из библиотеки Pandas. Метод `TSNE()` (из библиотеки *scikit-learn*) принимает два аргумента, которые мы должны рассмотреть внимательнее:

- `n_components` — число измерений, которое требуется получить, то есть если передать в нем число `2`, мы получим двумерный результат, а если передать `3` — трехмерный;
- `n_iter` — количество итераций по исходным данным. Так же как в алгоритме word2vec (см. листинг 11.8), итерации являются аналогами эпох обучения нейронной сети. Чем больше итераций, тем дольше будет продолжаться обучение и тем лучше могут получаться результаты (хотя только до определенного момента).

Листинг 11.9. Применение метода t-SNE для сокращения размерности пространства

```
tsne = TSNE(n_components=2, n_iter=1000)
X_2d = tsne.fit_transform(model.wv[model.wv.vocab])
coords_df = pd.DataFrame(X_2d, columns=['x','y'])
coords_df['token'] = model.wv.vocab.keys()
```

На выполнение кода t-SNE из листинга 11.9 может потребоваться некоторое время, поэтому, если вам не терпится перейти к следующим примерам, можете воспользоваться нашими результатами[2,3]:

```
coords_df = pd.read_csv('clean_gutenberg_tsne.csv')
```

---

[1] *van der Maaten L. & Hinton G.* (2008). «Visualizing data using t-SNE». Journal of Machine Learning Research, 9, 2579–605.

[2] Мы создали предлагаемый файл CSV, запустив алгоритм t-SNE с помощью этой команды: `ordins_df.to_csv ('clean_gutenberg_tsne.csv', index = False)`.

[3] Обратите внимание, что в силу стохастической природы алгоритма t-SNE вы будете получать уникальные результаты при каждом его запуске.

Независимо от того как вы получили результаты в `coords_df`, самостоятельно запустив алгоритм t-SNE или загрузив их из нашего файла, вы можете проверить первые несколько строк в объекте `DataFrame` с помощью его метода `head()`:

```
coords_df.head()
```

На рис. 11.7 показан результат, который мы получили вызовом `head()` у себя.

В листинге 11.10 показан код, создающий статическую диаграмму рассеяния (рис. 11.8) двумерных данных, созданных с помощью t-SNE (в листинге 11.9).

Листинг 11.10. Статическая двумерная диаграмма рассеяния векторного пространства слов

```
_ = coords_df.plot.scatter('x', 'y', figsize=(12,12),
                           marker='.', s=10, alpha=0.2)
```

	x	y	token
0	62.494060	8.023034	emma
1	8.142986	33.342200	by
2	62.507140	10.078477	jane
3	12.477635	17.998343	volume
4	25.736960	30.876250	i

Рис. 11.7. Это объект DataFrame из библиотеки Pandas с двумерным представлением векторного пространства слов, созданного на основе корпуса из Проекта Гутенберг. Каждая отдельная лексема имеет свои координаты x и y

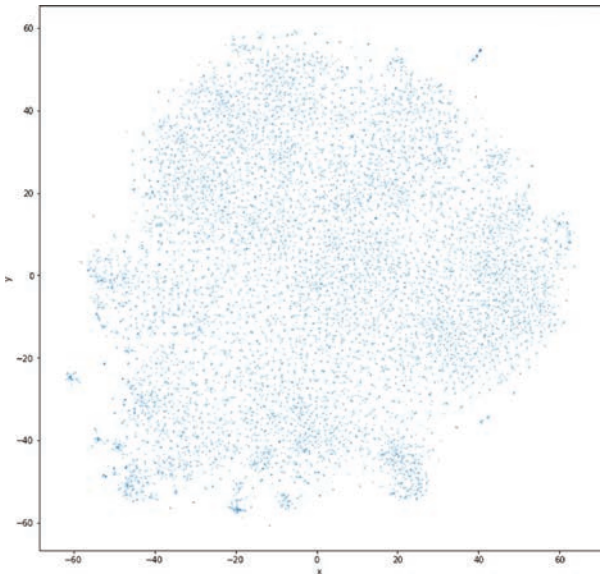


Рис. 11.8. Статическая двумерная диаграмма рассеяния векторного пространства слов

Диаграмма рассеяния, показанная на рис. 11.8, может выглядеть интересной, но из нее не удастся извлечь много полезной информации. Для создания более информативной и интерактивной диаграммы мы рекомендуем использовать библиотеку *bokeh*, как показано в листинге 11.11[1].

Листинг 11.11. Интерактивная диаграмма bokeh с двумерными данными из векторного пространства слов

```
output_notebook()
subset_df = coords_df.sample(n=5000)
p = figure(plot_width=800, plot_height=800)
_ = p.text(x=subset_df.x, y=subset_df.y, text=subset_df.token)
show(p)
```

Код в листинге 11.11 создает интерактивную диаграмму рассеяния, изображенную на рис. 11.9, используя координаты *x* и *y*, сгенерированные алгоритмом t-SNE.

Включив кнопку *Wheel Zoom* (Масштабировать колесиком мыши) в правом верхнем углу графика, вы сможете использовать мышь, чтобы увеличить масштаб выбранной области в облаке слов, и тогда слова станут разборчивыми.

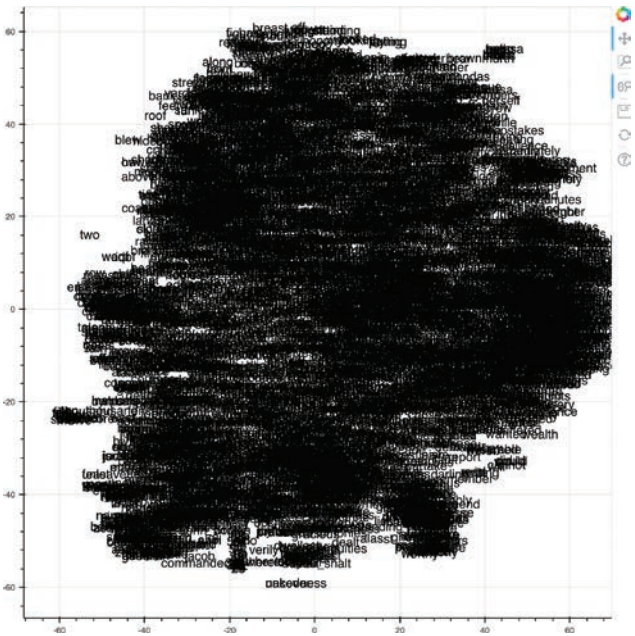


Рис. 11.9. Интерактивная диаграмма bokeh, отображающая двумерное векторное пространство слов

[1] В листинге 11.11 мы использовали метод `sample()` из библиотеки Pandas, чтобы сократить набор данных до 5000 лексем, потому что обнаружили, что при большем количестве данных взаимодействовать с интерактивной диаграммой, созданной библиотекой bokeh, становится неудобно.

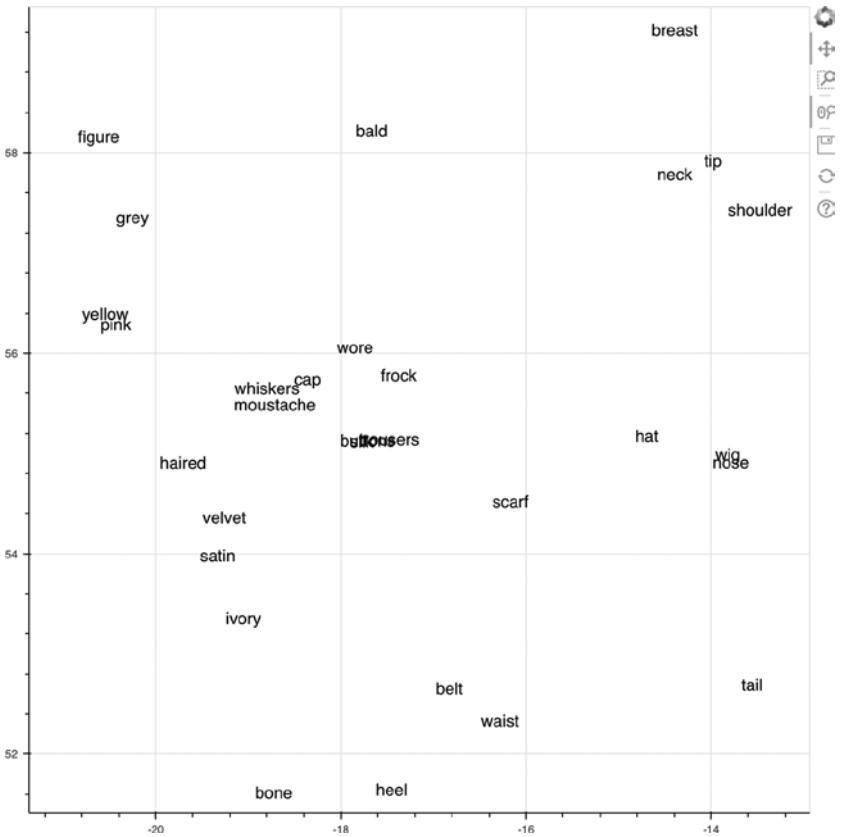


Рис. 11.10. Слова, связанные с понятием одежды, из корпуса Гутенберга, обнаруженные при увеличении области диаграммы bokeh из рис. 11.9

Например, как показано на рис. 11.10, мы выявили область, состоящую в основном из слов, обозначающих предметы одежды, с близкими к ним группами слов, относящихся к анатомии человека, цвету и типам тканей. Такое исследование позволяет оценить, хотя и субъективно, как объединяются родственные слова, особенно синонимы. В ходе такого исследования можно также подметить недостатки этапов предварительной обработки, как, например, включение знаков препинания, выделение биграмм или других лексем, которые было бы желательно не включать в словарь.

ПЛОЩАДЬ ПОД КРИВОЙ ROC

Приносим извинения за то, что приостанавливаем забавные эксперименты с интерактивными диаграммами векторов слов. Но мы вынуждены сделать

небольшой перерыв в обсуждении обработки естественного языка, чтобы представить метрику, которая пригодится нам в следующем разделе, где мы оценим качество моделей NLP глубокого обучения.

До сих пор большинство моделей, представленных в этой книге, были мультиклассовыми классификаторами: например, в моделях распознавания цифр из коллекции MNIST использовалось 10 выходных нейронов, представляющих вероятность для каждой из 10 возможных цифр, что именно она находится на входном изображении. Однако в остальных разделах этой главы наши модели глубокого обучения будут действовать как *бинарные классификаторы*, то есть они будут различать только два класса. В частности, мы создадим бинарные классификаторы, оценивающие эмоциональную окраску (положительную или отрицательную) отзывов к фильмам.

В отличие от искусственных нейронных сетей, решающих задачи мультиклассовой классификации, в которых число выходных нейронов должно совпадать с числом классов, сетям, действующим в роли бинарных классификаторов, достаточно одного выходного нейрона. Причина такого положения обусловлена отсутствием дополнительной информации, требующей наличия двух выходных нейронов. Если бинарный классификатор получает некоторый вход x и вычисляет некоторый выход $\hat{y}$ для одного из классов, то выход для другого класса равен разности $1 - \hat{y}$. Например, если в бинарный классификатор передается отзыв к фильму и тот выводит вероятность 0.85, что этот отзыв является положительным, тогда вероятность, что отзыв является отрицательным, равна $1 - 0.85 = 0.15$.

Поскольку бинарные классификаторы имеют один выход, для оценки качества модели можно использовать более сложные метрики, чем черно-белая метрика *точности*, преобладающая в задачах мультиклассовой классификации. Например, при типичном расчете точности можно утверждать, что если $\hat{y} > 0.5$, то модель предсказывает принадлежность входа x одному, а если меньше 0.5 — то другому классу. Чтобы понять, почему использование определенного бинарного порога, подобного этому, считается чрезмерно упрощенным подходом, рассмотрим ситуацию, когда при вводе отзыва к фильму получается прогноз $\hat{y} = 0.48$: согласно типичному подходу с использованием порогового значения, можно утверждать, что поскольку $\hat{y}$ меньше 0.5, этот отзыв должен классифицироваться как отрицательный. Если второму отзыву соответствует выходное значение $\hat{y} = 0.51$, модель едва ли более уверена, что он является положительным по сравнению с первым отзывом. Тем не менее, поскольку 0.51 превышает порог точности 0.5, второй отзыв классифицируется как положительный.

Строгость метрики порогового значения может скрыть немало нюансов в выходных данных модели, поэтому при оценке качества бинарных классификаторов мы предпочитаем метрику, называемую *площадью под кривой рабочей характеристики приемника* (Area Under the Curve of the Receiver Operating Characteristic — ROC AUC). Использование площади под кривой ROC в роли метрики началось еще со времен Второй мировой войны — в ту пору она была

разработана для оценки суждений операторов радаров, когда они пытались идентифицировать присутствие объектов противника.

ROC AUC нам нравится по двум причинам.

1. Он объединяет две полезные метрики — *количество истинно-положительных* и *ложно-положительных решений* — в одно суммарное значение.
2. Позволяет оценить качество бинарного классификатора по всему диапазону $\hat{y}$, от 0.0 до 1.0. Это контрастирует с метрикой точности, которая оценивает качество двоичного классификатора только по одному пороговому значению — обычно $\hat{y} = 0{:}\ 5$.

МАТРИЦА ОШИБОК

Чтобы понять, как рассчитать метрику ROC AUC, прежде нужно разобраться с так называемой *матрицей ошибок*, которая, как вы увидите, не так уж и сложна. Она представляет простую таблицу 2 × 2, описывающую, сколько ошибок допустила модель (или, как во времена Второй мировой войны, человек), действующая как бинарный классификатор. Пример матрицы можно увидеть в табл. 11.3.

Таблица 11.3. Матрица ошибок

		Фактическое значение *y*	
		1	**0**
Предсказанное значение *y*	**1**	Истинно-положительный результат	Ложно-положительный результат
	0	Ложно-отрицательный результат	Истинно-отрицательный результат

Чтобы наполнить матрицу ошибок конкретикой, вернемся к бинарному классификатору хот-дог/не хот-дог, который мы использовали в качестве примера во многих предыдущих главах.

- Когда мы вводим в модель некоторые данные x и она *предсказывает*, что эти данные представляют хот-дог, мы имеем дело с первой строкой таблицы, где *предсказанное значение* $y = 1$. В этом случае:
  - результат считается *истинно-положительным*, если на вход была передана информация *действительно* о хот-доге (фактическое значение $y = 1$) и модель правильно классифицировала данные;
  - результат считается *ложно-положительным*, если в действительности на вход была передана информация *не* о хот-доге (фактическое значение $y = 0$) и модель *ошиблась*.

- Когда мы вводим в модель некоторые данные x и она *предсказывает*, что эти данные *не* представляют хот-дог, мы имеем дело со второй строкой таблицы, где *предсказанное значение* $y = 1$. В этом случае:
  - результат считается *ложно-отрицательным*, если на вход была передана информация *действительно* о хот-доге (фактическое значение $y = 1$) и модель ошиблась;
  - результат считается *истинно-отрицательным*, если в действительности на вход была передана информация *не* о хот-доге (фактическое значение $y = 0$) и модель правильно классифицировала ее.

ВЫЧИСЛЕНИЕ МЕТРИКИ ROC AUC

Познакомившись с матрицей ошибок, мы можем двинуться дальше и рассчитать саму метрику ROC AUC, взяв за образец игрушечный пример. Допустим, как показано в табл. 11.4, мы передали в модель бинарной классификации четыре набора входных данных. Два из них представляют хот-доги ($y = 1$), а два — нет ($y = 0$). Для каждого из этих наборов модель выводит одно из прогнозных значений $\hat{y}$, которые также перечислены в табл. 11.4.

Таблица 11.4. Четыре предсказания хот-дог/не хот-дог

y	$\hat{y}$
0	0.3
1	0.5
0	0.6
1	0.9

Чтобы вычислить метрику ROC AUC, рассмотрим каждое из значений $\hat{y}$, возвращаемых моделью, как порог бинарной классификации. Начнем с самого низкого значения $\hat{y}$, равного 0.3 (см. столбец «Порог 0.3» в табл. 11.5). При этом пороге только первый входной набор в табл. 11.4 классифицируется как *не* хот-дог, тогда как наборы со второго по четвертый (все с $\hat{y} > 0.3$) классифицируются как хот-доги. Мы можем сравнить каждое из этих четырех предсказаний с матрицей ошибок в табл. 11.3:

1. **Истинно-отрицательный результат (ИО)**: этот набор данных представляет нечто иное, не являющееся хот-догом ($y = 0$), и был правильно классифицирован.
2. **Истинно-положительный результат (ИП)**: этот набор данных представляет хот-дог ($y = 1$) и был правильно классифицирован.

3. **Ложно-положительный результат (ЛП)**: этот набор данных представляет нечто иное, не являющееся хот-догом ($y = 0$), но был классифицирован неправильно.
4. **Истинно-положительный результат (ИП)**: этот набор данных, как и набор 2, представляет хот-дог ($y = 1$) и был правильно классифицирован.

Таблица 11.5. Четыре предсказания хот-дог/не хот-дог с промежуточными вычислениями ROC AUC

y	$\hat{y}$	Порог 0.3	Порог 0.5	Порог 0.6
0 (не хот-дог)	0.3	0 (ИО)	0 (ИО)	0 (ИО)
1 (хот-дог)	0.5	1 (ИП)	0 (ЛО)	0 (ЛО)
0 (не хот-дог)	0.6	1 (ЛП)	1 (ЛП)	0 (ИО)
1 (хот-дог)	0.9	1 (ИП)	1 (ИП)	1 (ИП)
Доля истинно-положительных результатов $= \frac{\text{ИП}}{\text{ИП} + \text{ЛО}}$		$\frac{2}{2+0} = 1.0$	$\frac{1}{1+1} = 0.5$	$\frac{1}{1+1} = 0.5$
Доля ложно-положительных результатов $= \frac{\text{ЛП}}{\text{ЛП} + \text{ИО}}$		$\frac{1}{1+1} = 0.5$	$\frac{1}{1+1} = 0.5$	$\frac{0}{0+2} = 0.0$

Те же вычисления повторяются для порога 0.5 и затем для порога 0.6, чтобы заполнить оставшиеся столбцы в табл. 11.5. Проработайте эти два столбца самостоятельно, сравнивая результаты классификации для каждого порога с фактическими значениями y и матрицей ошибок (табл. 11.3), чтобы лучше разобраться в этих понятиях. Наконец, обратите внимание, что наибольшее значение $\hat{y}$ (в данном случае 0.9) можно пропустить и не рассматривать как потенциальный порог, потому что при таком высоком пороге все четыре образца будут классифицированы как не хот-доги, что делает его потолком, а не границей классификации.

Следующий шаг на пути к получению метрики ROC AUC — вычисление доли истинно-положительных (True Positive Rate, TPR) и ложно-положительных (False Positive Rate, FPR) результатов для каждого из трех порогов. В уравнениях 11.1 и 11.2 используется столбец «Порог 0.3», чтобы наглядно предоставить порядок вычисления долей истинно-положительных и ложно-положительных результатов соответственно.

$$\text{Доля истинно-положительных результатов} = \\ = \frac{(\text{количество ИП})}{(\text{количество ИП}) + (\text{количество ЛО})} = \frac{2}{2+0} = \frac{2}{2} = 1.0. \qquad (11.1)$$

$$\text{Доля ложно-положительных результатов} = \\ = \frac{(\text{количество ЛП})}{(\text{количество ЛП}) + (\text{количество ИО})} = \frac{1}{1+1} = \frac{1}{2} = 0. \quad (11.2)$$

Вычисления долей TPR и FPR для порогов 0.5 и 0.6 для удобства представлены в нижней части табл. 11.5. И снова проверьте эти вычисления вручную, чтобы впоследствии, когда это понадобится, вы смогли выполнить их самостоятельно.

Завершающий этап в вычислении метрики ROC AUC — создание графика, подобного изображенному на рис. 11.11. Точками, образующими кривую рабочей характеристики приемника (Receiver Operating Characteristic, ROC), являются доли ложно-положительных (горизонтальная координата x) и истинно-положительных (вертикальная координата y) результатов для каждого из имеющихся порогов (в данном случае у нас их три) в табл. 11.5 плюс две дополнительные точки в левом нижнем и правом верхнем углах графика. Вот эти пять точек (показаны оранжевым цветом на рис. 11.11):

1. (0, 0) — левый нижний угол.
2. (0, 0.5) — порог 0.6.
3. (0.5, 0.5) — порог 0.5.
4. (0.5, 1) — порог 0.3.
5. (1, 1) — правый верхний угол.

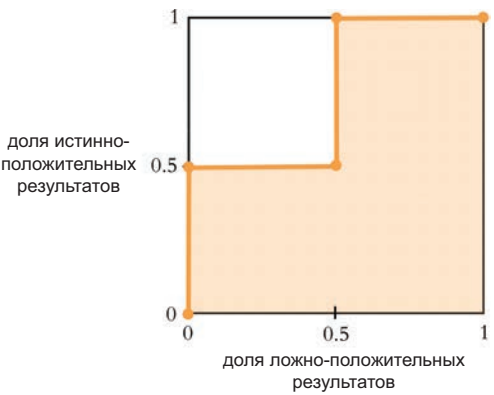


Рис. 11.11. Площадь под кривой (закрашена оранжевым) рабочей характеристики приемника определяется вычислением долей истинно-положительных и ложно-положительных результатов, как показано в табл. 11.5

В этом игрушечном примере мы использовали только четыре значения $\hat{y}$, поэтому получили всего пять точек, определяющих форму кривой ROC, из-за чего она обрела ступенчатый вид. При наличии большого числа прогнозов, предоставляющих много разных значений $\hat{y}$, что не редкость в реальных примерах, кривая ROC образуется большим числом точек и имеет гораздо более гладкую форму, скорее напоминая собственно кривую. Площадь под кривой (Area Under

the Curve, AUC) ROC — это именно то, что означают эти слова: на рис. 11.11 мы закрасили эту область оранжевым цветом, и в данном примере AUC составляет 75% от всей области графика, поэтому метрика ROC AUC равна 0.75.

Бинарный классификатор, который работает не лучше, чем метод случайного угадывания, сгенерирует прямую диагональ, идущую из левого нижнего угла графика в правый верхний, то есть метрика ROC AUC, равная 0.5, указывает, что классификатор работает ничуть не лучше метода, основанного на подбрасывании монеты. В идеальном случае метрика ROC AUC равна 1.0, то есть когда FPR = 0 и TPR = 1 для всех имеющихся пороговых значений $\hat{y}$. При разработке бинарного классификатора с хорошей метрикой ROC AUC целью является минимизация FPR и максимизация TPR в диапазоне порогов $\hat{y}$. Однако в большинстве задач, с которыми вы столкнетесь, достижение идеальной метрики ROC AUC, равной 1.0, невозможно: обычно в данных присутствует некоторый шум — иногда много шума, — который не позволяет достичь совершенства. Для любого набора данных существует некоторое (обычно неизвестное!) максимальное значение метрики ROC AUC, и независимо от того, насколько идеально ваша модель справляется с ролью бинарного классификатора, есть определенный потолок ROC AUC, который не сможет преодолеть ни одна модель.

В оставшейся части этой главы мы используем метрику ROC AUC, а также некоторые более простые метрики точности и стоимости, с которыми вы уже знакомы, для оценки качества моделей глубокого обучения бинарной классификации, которые будем разрабатывать и обучать.

КЛАССИФИКАЦИЯ ЕСТЕСТВЕННОГО ЯЗЫКА С ИСПОЛЬЗОВАНИЕМ УЖЕ ЗНАКОМЫХ СЕТЕЙ

В этом разделе мы объединим идеи, представленные выше — лучшие методы предварительной обработки естественного языка, создание векторов слов и метрику ROC AUC, — с теорией глубокого обучения из предыдущих глав. Как уже упоминалось, модель обработки естественного языка, с которой мы будем экспериментировать в оставшейся части главы, является бинарным классификатором, который предсказывает эмоциональную окраску, положительную или отрицательную, отзывов к фильмам. Начнем с классификации документов на естественном языке с использованием уже знакомых типов нейронных сетей — полносвязанных и сверточных, — а затем перейдем к сетям, специализирующимся на обработке последовательных данных.

ЗАГРУЗКА ОТЗЫВОВ К ФИЛЬМАМ ИЗ IMDB

Чтобы получить базовый показатель качества, на который потом можно будет ориентироваться, мы сначала обучим и протестируем относительно простую

полносвязанную сеть. Весь код этого примера вы найдете в нашем блокноте Jupyter *dense_sentiment_classifier.ipynb*.

В листинге 11.12 перечислены зависимости, которые понадобятся нашему полносвязанному классификатору эмоциональной окраски. Многие из этих зависимостей уже знакомы вам по предыдущим главам, но среди них есть и новые (например, набор данных с отзывами к фильмам, средства для сохранения параметров модели во время обучения, расчет ROC AUC). Как обычно, подробнее эти зависимости будут рассматриваться позже, по мере их применения.

Листинг 11.12. Загрузка зависимостей для классификатора эмоциональной окраски

```
import keras
from keras.datasets import imdb # new!
from keras.preprocessing.sequence import pad_sequences # new!
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.layers import Embedding # новая!
from keras.callbacks import ModelCheckpoint # новая!
import os # новая!
from sklearn.metrics import roc_auc_score, roc_curve # новая!
import pandas as pd
import matplotlib.pyplot as plt # новая!
%matplotlib inline
```

Хорошей практикой считается размещение как можно большего числа гиперпараметров в верхней части файла. Это упростит эксперименты с ними. Это также поможет вам (или вашим коллегам) лучше понять, что делает файл, когда вы вернетесь к нему (возможно, намного) позже. Поэтому поместим все наши гиперпараметры в одну ячейку в блокноте Jupyter, как показано в листинге 11.13.

Листинг 11.13. Настройка параметров классификатора эмоциональной окраски

```
# имя каталога для сохранения результатов:
output_dir = 'model_output/dense'

# обучение:
epochs = 4
batch_size = 128

# параметры векторного пространства слов:
n_dim = 64
n_unique_words = 5000
n_words_to_skip = 50
max_review_length = 100
pad_type = trunc_type = 'pre'

# архитектура полносвязанной сети:
n_dense = 64
dropout = 0.5
```

Рассмотрим поближе каждую из этих переменных:

- `output_dir`: имя каталога (в идеале уникальное), куда будут сохраняться параметры модели после каждой эпохи обучения, чтобы позже можно было вернуться к параметрам из любой эпохи.
- `epochs`: количество эпох обучения модели; модели NLP обычно достигают стадии переобучения за меньшее количество эпох, чем модели компьютерного зрения.
- `batch_size`: как и прежде, эта переменная определяет число обучающих образцов, которое должно быть передано в модель в каждом цикле обучения (см. рис. 8.5).
- `n_dim`: число измерений в векторном пространстве слов.
- `n_unique_words`: описывая алгоритм word2vec выше в этой главе, мы включали в словарь векторного пространства только лексемы, которые встречаются в корпусе не менее определенного числа раз. Альтернативный подход, который мы предлагаем здесь, состоит в том, чтобы отсортировать все лексемы в корпусе по количеству вхождений, а затем использовать только определенное количество самых популярных слов. Эндрю Маас и его коллеги[1] решили использовать в своем эксперименте 5000 самых популярных слов, встречающихся в отзывах к фильмам, и мы поступим так же[2].
- `n_words_to_skip`: Маас с коллегами не стали составлять список стоп-слов вручную и предположили, что 50 наиболее часто встречающихся слов в корпусе могут служить достойными представителями на роль стоп-слов. Мы последовали их примеру и сделали то же самое[3].
- `max_review_length`: все отзывы должны иметь одинаковую длину, чтобы мы могли сообщить библиотеке TensorFlow форму входных данных для нашей модели глубокого обучения. В данном случае мы решили ограничить длину отзывов 100 словами[4]. Любые отзывы, содержащие более 100 слов, будут усекаться, а отзывы короче 100 слов — дополняться специальным *символом заполнения* (по аналогии с заполнением нулями в задачах компьютерного зрения, как показано на рис. 10.3).
- `pad_type`: если присвоить этой переменной строку `'pre'`, символы заполнения будут добавляться в начало каждого короткого отзыва. Также этой

---

[1] Выше в этой главе мы уже упоминали работу Мааса и его коллег (2011). Они собрали корпус отзывов к фильмам, который мы используем в своем блокноте.

[2] Порог в 5000 слов может быть неоптимальным, но мы не стали тратить время на тестирование других значений. Вы можете сделать это сами!

[3] Еще раз обратите внимание, что следование примеру Мааса и его коллег может быть не лучшим выбором. Также это означает, что мы фактически включим в словарь с 51-го наиболее популярного слова по 5050-е.

[4] Вы можете поэкспериментировать с более длинными или короткими отзывами.

переменной можно присвоить строку `'post'`, и тогда заполняющие символы будут добавляться в конец. Для полносвязанной сети, как в этом блокноте, не имеет большого значения, какой из вариантов выбрать. Но далее в этой главе, когда мы перейдем к специализированным типам слоев для обработки последовательных данных[1], лучше будет использовать `'pre'`, потому что содержимое в конце документа оказывает большее влияние на решение, принимаемое моделью, и поэтому неинформативные заполняющие символы лучше добавлять в начало документа.

- `trunc_type`: по аналогии с `pad_type`, этой переменной можно присвоить строку `'pre'` или `'post'`. В первом случае длинные отзывы будут усекаться с начала, а во втором — с конца. Выбирая `'pre'`, мы делаем (смелое!) предположение, что в конце отзывов к фильмам, как правило, содержится больше информации о настроении рецензента, чем в начале.
- `n_dense`: количество нейронов в полносвязанном слое нашей нейронной сети. Мы наобум выбрали число 64, поэтому, если хотите, можете поэкспериментировать и попытаться оптимизировать этот гиперпараметр. Также для простоты мы используем один полносвязанный слой, но вы можете попробовать добавить еще несколько.
- `dropout`: определяет, сколько нейронов будет отключаться в полносвязанном слое. И снова мы не стали тратить время на оптимизацию этого гиперпараметра (выбрав его равным 0.5).

Загрузка набора данных с отзывами к фильмам выполняется одной строкой кода, как показано в листинге 11.14.

Листинг 11.14. Загрузка отзывов к фильмам из корпуса IMDb

```
(x_train, y_train), (x_valid, y_valid) = \
    imdb.load_data(num_words=n_unique_words, skip_top=n_words_to_skip)
```

Этот набор данных, созданный Маасом и его коллегами (2011), состоит из отзывов на естественном языке к фильмам, взятых из общедоступной базы данных Internet Movie Database (IMDb; *imdb.com*). Он включает 50 000 отзывов, половина из которых находится в обучающем наборе данных (`x_train`), а половина — в тестовом (`x_valid`). Вместе с отзывами пользователи указывают рейтинг фильма по 10-балльной шкале. Метки (`y_train` и `y_valid`) имеют бинарный характер и назначены в соответствии с этими 10-балльными оценками:

- отзывы с четырьмя звездами или меньше считаются отрицательным ($y = 0$);
- отзывы с семью звездами или больше считаются положительными ($y = 1$);
- умеренные отзывы — с пятью или шестью звездами — не включались в набор данных, что упрощает задачу бинарной классификации.

[1] Например, RNN, LSTM.

С помощью аргументов `num_words` и `skip_top` в вызове `imdb.load_data()` мы ограничиваем размер словаря и удаляем наиболее распространенные слова (стоп-слова) соответственно.

В нашем блокноте *dense_sentiment_classifier.ipynb* использована удобная возможность загрузки данных с отзывами к фильмам из IMDb с помощью метода `imdb.load_data()` из библиотеки Keras. При работе с собственными данными на естественном языке вам, вероятно, потребуется применить некоторые виды предварительной обработки. В дополнение к общему руководству по предварительной обработке, которое мы представили выше в этой главе, Keras предоставляет ряд удобных утилит предварительной обработки текстовых данных, которые описаны в онлайн-документации по адресу *keras.io/preprocessing/text*. В частности, класс `Tokenizer()` позволяет выполнить все необходимые этапы предварительной обработки одной строкой кода, включая:

- лексемизацию корпуса на уровне слов (и даже на уровне символов);
- настройку размера словаря для векторного пространства слов (значением `num_words`);
  - удаление знаков препинания;
  - преобразование символов в нижний регистр;
  - преобразование лексем в целочисленные индексы.

ИССЛЕДОВАНИЕ ДАННЫХ ИЗ IMDB

Выполнив команду `x_train[0:6]`, можно исследовать первые шесть отзывов из обучающего набора данных, первые два из которых показаны на рис. 11.12. Эти отзывы изначально представлены в формате целочисленных индексов, где каждая уникальная лексема из набора данных представлена целым числом. Первые несколько целых чисел — особый случай, в соответствии с соглашениями, которые широко используются в обработке естественного языка:

- `0`: зарезервирован для представления заполняющих лексем (которые мы будем добавлять в отзывы, содержащие меньше `max_review_length` слов).
- `1`: будет играть роль *начальной лексемы*, отмечающей начало отзыва. Однако, как указано в следующем пункте, начальная лексема входит в число 50 самых распространенных лексем и поэтому отображается как `UNK` (неизвестно).
- `2`: любые лексемы, частотные (входящие в число 50 самых распространенных слов) или редкие (находящиеся ниже 5050 самых распространенных слов), окажутся за пределами нашего словаря и, соответственно, будут заменяться этим числом, обозначающим *неизвестную лексему*.
- `3`: слово, наиболее часто встречающееся в корпусе.
- `4`: слово, занимающее вторую позицию по числу вхождений в корпус.
- `5`: слово, занимающее третью позицию по числу вхождений в корпус, и т. д.

Выполнив код из листинга 11.15, можно узнать длину первых шести отзывов в обучающем наборе данных.

Листинг 11.15. Вывод числа лексем в шести отзывах

```
for x in x_train[0:6]:
    print(len(x))
```

Как видите, отзывы имеют разную длину, от 43 до 550 лексем. Вскоре мы исправим это непостоянство, приведя их к одинаковой длине.

Отзывы к фильмам вводятся в нашу нейронную сеть в формате целочисленных индексов, как показано на рис. 11.12, потому что это наиболее эффективный формат хранения информации о лексемах. Например, для передачи лексем в виде символьных строк потребуется значительно больше памяти. Однако нам, людям, не особенно интересно рассматривать отзывы в формате целочисленных индексов. Чтобы преобразовать их обратно в естественный язык, создадим каталог слов, как показано ниже, где `PAD`, `START` и `UNK` используются для представления заполняющей, начальной и неизвестной лексемы соответственно:

```
word_index = keras.datasets.imdb.get_word_index()
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["PAD"] = 0
word_index["START"] = 1
word_index["UNK"] = 2
index_word = {v:k for k,v in word_index.items()}
```

```
array([ [2, 2, 2, 2, 2, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 2,
173, 2, 256, 2, 2, 100, 2, 838, 112, 50, 670, 2, 2, 2, 480, 284, 2, 150,
2, 172, 112, 167, 2, 336, 385, 2, 2, 172, 4536, 1111, 2, 546, 2, 2, 447,
2, 192, 50, 2, 2, 147, 2025, 2, 2, 2, 2, 1920, 4613, 469, 2, 2, 71, 87,
2, 2, 2, 530, 2, 76, 2, 2, 1247, 2, 2, 2, 515, 2, 2, 2, 626, 2, 2, 2, 62,
386, 2, 2, 316, 2, 106, 2, 2, 2223, 2, 2, 480, 66, 3785, 2, 2, 130, 2, 2,
2, 619, 2, 2, 124, 51, 2, 135, 2, 2, 1415, 2, 2, 2, 2, 215, 2, 77, 52, 2,
2, 407, 2, 82, 2, 2, 2, 107, 117, 2, 2, 256, 2, 2, 2, 3766, 2, 723, 2, 7
1, 2, 530, 476, 2, 400, 317, 2, 2, 2, 2, 1029, 2, 104, 88, 2, 381, 2, 29
7, 98, 2, 2071, 56, 2, 141, 2, 194, 2, 2, 2, 226, 2, 2, 134, 476, 2, 480,
2, 144, 2, 2, 2, 51, 2, 2, 224, 92, 2, 104, 2, 226, 65, 2, 2, 1334, 88,
2, 2, 283, 2, 2, 4472, 113, 103, 2, 2, 2, 2, 2, 178, 2],
       [2, 194, 1153, 194, 2, 78, 228, 2, 2, 1463, 4369, 2, 134, 2, 2, 71
5, 2, 118, 1634, 2, 394, 2, 2, 119, 954, 189, 102, 2, 207, 110, 3103, 2,
2, 69, 188, 2, 2, 2, 2, 2, 249, 126, 93, 2, 114, 2, 2300, 1523, 2, 647,
2, 116, 2, 2, 2, 2, 229, 2, 340, 1322, 2, 118, 2, 2, 130, 4901, 2, 2, 100
2, 2, 89, 2, 952, 2, 2, 2, 455, 2, 2, 2, 2, 1543, 1905, 398, 2, 1649, 2,
2, 2, 163, 2, 3215, 2, 2, 1153, 2, 194, 775, 2, 2, 2, 349, 2637, 148, 60
5, 2, 2, 2, 123, 125, 68, 2, 2, 2, 349, 165, 4362, 98, 2, 2, 228, 2, 2,
2, 1157, 2, 299, 120, 2, 120, 174, 2, 220, 175, 136, 50, 2, 4373, 228, 2,
2, 2, 656, 245, 2350, 2, 2, 2, 131, 152, 491, 2, 2, 2, 2, 1212, 2, 2, 2,
371, 78, 2, 625, 64, 1382, 2, 2, 168, 145, 2, 2, 1690, 2, 2, 2, 1355, 2,
2, 2, 52, 154, 462, 2, 89, 78, 285, 2, 145, 95],
```

Рис. 11.12. Первые два отзыва из обучающего набора данных, созданного Эндрю Маасом и его коллегами (2011) на основе базы данных IMDb. Лексемы представлены в формате целочисленных индексов

После этого можно использовать код из листинга 11.16 для просмотра любого отзыва фильма по вашему выбору — в данном случае выводится текст первого отзыва из обучающего набора.

Листинг 11.16. Вывод отзыва в виде строки символов

```
' '.join(index_word[id] for id in x_train[0])
```

Получившаяся строка должна выглядеть в точности как показано на рис. 11.13.

```
"UNK UNK UNK UNK UNK brilliant casting location scenery story direction e
veryone's really suited UNK part UNK played UNK UNK could UNK imagine bei
ng there robert UNK UNK UNK amazing actor UNK now UNK same being director
UNK father came UNK UNK same scottish island UNK myself UNK UNK loved UNK
fact there UNK UNK real connection UNK UNK UNK UNK witty remarks througho
ut UNK UNK were great UNK UNK UNK brilliant UNK much UNK UNK bought UNK U
NK UNK soon UNK UNK UNK released UNK UNK UNK would recommend UNK UNK ever
yone UNK watch UNK UNK fly UNK UNK amazing really cried UNK UNK end UNK U
NK UNK sad UNK UNK know what UNK say UNK UNK cry UNK UNK UNK UNK must UNK
been good UNK UNK definitely UNK also UNK UNK UNK two little UNK UNK play
ed UNK UNK UNK norman UNK paul UNK were UNK brilliant children UNK often
left UNK UNK UNK UNK list UNK think because UNK stars UNK play them UNK g
rown up UNK such UNK big UNK UNK UNK whole UNK UNK these children UNK ama
zing UNK should UNK UNK UNK what UNK UNK done don't UNK think UNK whole s
tory UNK UNK lovely because UNK UNK true UNK UNK someone's life after UNK
UNK UNK UNK UNK us UNK"
```

Рис. 11.13. Первый отзыв к фильму из обучающего набора, теперь в виде строки символов

Напомню, что отзыв на рис. 11.13 содержит лексемы, которые передаются в нейронную сеть, однако мы сами, скорее всего, предпочли бы прочитать полный отзыв, без всех этих лексем `UNK`. Иногда с целью отладки модели может быть даже полезно видеть полные отзывы. Например, если мы проявим излишнюю жесткость или осмотрительность в отношении настройки порогов `n_unique_words` или `n_words_to_skip`, это может стать очевидным при сравнении отзыва, подобного приведенному на рис. 11.13, с полным текстом. Для этого достаточно просто загрузить полные отзывы:

```
(all_x_train,_),(all_x_valid,_) = imdb.load_data()
```

и изменить листинг 11.16, чтобы применить `join()` к полному отзыву по нашему выбору (`all_x_train` или `all_x_valid`), как показано в листинге 11.17.

Листинг 11.17. Вывод полного отзыва в виде строки символов

```
' '.join(index_word[id] for id in all_x_train[0])
```

Этот код выведет полный текст выбранного вами отзыва — в данном случае первого в обучающем наборе, как показано на рис. 11.14.

```
"START this film was just brilliant casting location scenery story direct
ion everyone's really suited the part they played and you could just imag
ine being there robert redford's is an amazing actor and now the same bei
ng director norman's father came from the same scottish island as myself
so i loved the fact there was a real connection with this film the witty
remarks throughout the film were great it was just brilliant so much that
i bought the film as soon as it was released for retail and would recomme
nd it to everyone to watch and the fly fishing was amazing really cried a
t the end it was so sad and you know what they say if you cry at a film i
t must have been good and this definitely was also congratulations to the
two little boy's that played the part's of norman and paul they were just
brilliant children are often left out of the praising list i think becaus
e the stars that play them all grown up are such a big profile for the wh
ole film but these children are amazing and should be praised for what th
ey have done don't you think the whole story was so lovely because it was
true and was someone's life after all that was shared with us all"
```

Рис. 11.14. Первый отзыв к фильму из обучающего набора, теперь в своем исходном виде

СТАНДАРТИЗАЦИЯ ДЛИН ОТЗЫВОВ

Выполнив код из листинга 11.15, мы обнаружили, что отзывы к фильмам имеют разную длину. Чтобы запустить модель, использующую библиотеку TensorFlow, нужно указать размер входных данных, поступающих в модель во время обучения. Это позволит TensorFlow оптимизировать распределение памяти и вычислительных ресурсов. Библиотека Keras предлагает для этого удобный метод `pad_sequence()`, дополняющий и обрезающий текстовые документы до определенного размера. Здесь мы стандартизируем обучающие и проверочные данные, как показано в листинге 11.18.

Листинг 11.18. Стандартизация длин отзывов дополнением и усечением

```
x_train = pad_sequences(x_train, maxlen=max_review_length,
                        padding=pad_type, truncating=trunc_type, value=0)
x_valid = pad_sequences(x_valid, maxlen=max_review_length,
                        padding=pad_type, truncating=trunc_type, value=0)
```

```
'PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD
PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD P
AD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PA
D PAD PAD UNK begins better than UNK ends funny UNK UNK russian UNK crew
UNK UNK other actors UNK UNK those scenes where documentary shots UNK UNK
spoiler part UNK message UNK UNK contrary UNK UNK whole story UNK UNK doe
s UNK UNK UNK UNK'
```

Рис. 11.15. Шестой отзыв из обучающего набора, дополненный лексемой PAD в начале до выбранной длины 100 лексем

Теперь, если вывести любой отзыв (например, обращением к `x_train[0:6]`) или его длину (например, выполнив код из листинга 11.15), можно увидеть, что все отзывы имеют одинаковую длину 100 (потому что мы установили `max_review_length = 100`). Исследовав отзыв `x_train[5]`, который прежде включал всего

43 лексемы, с помощью кода из листинга 11.16, можно заметить, что в начало отзыва было добавлено 57 лексем PAD (рис. 11.15).

ПОЛНОСВЯЗАННАЯ СЕТЬ

Обсудив теорию NLP, а также загрузив данные и выполнив их предварительную обработку, мы, наконец, готовы перейти к обсуждению архитектуры нейронной сети для классификации эмоциональной окраски отзывов. Модель полносвязанной сети для этой задачи, которая послужит нам точкой отсчета, показана в листинге 11.19.

Листинг 11.19. Полносвязанная архитектура для классификации эмоциональной окраски отзывов

```
model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(Flatten())
model.add(Dense(n_dense, activation='relu'))
model.add(Dropout(dropout))
# model.add(Dense(n_dense, activation='relu'))
# model.add(Dropout(dropout))
model.add(Dense(1, activation='sigmoid'))
```

Рассмотрим определение этой архитектуры строку за строкой:

- Вызовом метода `Sequential()` из библиотеки Keras мы создаем последовательную модель, как делали это во всех примерах в этой книге.
- По аналогии с алгоритмом word2vec, слой `Embedding()` позволяет создавать векторы слов из корпуса документов, в данном случае 25 000 отзывов к фильмам, образующего обучающий набор данных IMDb. В отличие от независимого создания векторов слов с помощью word2vec (или других алгоритмов, таких как GloVe), как мы делали выше в этой главе, обучение векторов слов с использованием обратного распространения в составе более широкой модели NLP имеет свои преимущества: координаты, которые получают слова в векторном пространстве, отражают не только сходство слов, но и конкретную конечную цель модели (например, бинарная классификация отзывов по эмоциональной окраске). Размер словаря и количество измерений векторного пространства определяются с помощью аргументов `n_unique_words` и `n_dim` соответственно. Поскольку слой `Embedding` является первым скрытым слоем сети, ему также следует передать форму входного слоя: это делается с помощью аргумента `input_length`.
- Как рассказывалось в главе 10, слой `Flatten()` позволяет передать многомерные данные (в данном случае двумерный выход слоя `Embedding`) в одномерный полносвязанный слой.

- Далее следует полносвязанный слой `Dense()`. В этой архитектуре мы использовали один такой слой, состоящий из нейронов с функцией активации `relu`, к которому добавили слой прореживания `Dropout()`.
- Мы выбрали очень неглубокую архитектуру для нейронной сети нашей базовой модели, но вы легко сможете углубить ее, добавив дополнительные слои `Dense()` (см. закомментированные строки в листинге 11.19).

Наконец, поскольку классификация осуществляется всего в два класса, нам нужен только один выходной нейрон (потому что, как обсуждалось выше в этой главе, если один класс имеет вероятность p, то другой — $1 - p$). В данном случае используется нейрон с функцией активации `sigmoid`, потому что он должен выводить величину вероятности в диапазоне от 0 до 1 (см. рис. 6.9).

Вместо создания векторного пространства слов на данных естественного языка (например, с помощью алгоритма word2vec или GloVe) или обучения слоя `Embedding` как части модели глубокого обучения предварительно обученное векторное пространство можно также загрузить из Интернета.

По аналогии со сверточной сетью, обученной на миллионах изображений из набора ImageNet (глава 10), перенос обучения на данных естественного языка является мощной возможностью. Эти векторы слов могли быть обучены на очень больших корпусах (например, на содержимом всей Википедии или всего англоязычного сегмента Интернета), представляющих обширные словари с множеством нюансов, обучать которые самостоятельно было бы слишком дорого. Примеры предварительно обученных векторных пространств слов доступны на github.com/Kyubyong/wordvectors и nlp.stanford.edu/projects/glove. Библиотека fastText также предлагает векторные представления слов на 157 языках; их можно загрузить с сайта fasttext.cc.

В этой книге мы не будем рассматривать использование предварительно обученных векторных пространств (загруженных или обученных отдельно от нашей модели глубокого обучения, как мы делали это выше в этой главе, вызывая `Word2Vec()`) вместо слоя `Embedding`, потому что сделать это можно слишком многими разными способами. Но если у вас появится желание пойти таким путем, прочитайте отличное руководство от Франсуа Шоле (François Chollet), создателя Keras, доступное по адресу bit.ly/preTrained.

Вызвав `model.summary()`, можно обнаружить, что эта очень простая модель NLP имеет довольно много параметров, как показано на рис. 11.16:

- Слой `Embedding` со словарем в 5000 слов, каждому из которых назначается 64 координаты в векторном пространстве, имеет 320 000 параметров ($64 \times \times 5000 = 320\,000$).
- Из слоя `Embedding` через слой `Flatten` в полносвязанный слой `Dense` передается 6400 значений: каждый отзыв состоит из 100 лексем, и каждая лексема определяется 64 координатами в векторном пространстве ($64 \times 100 = 6400$).

- Каждый из 64 нейронов в полносвязанном слое `Dense` получает на входе 6400 значений, исходящих из слоя `Flatten`, то есть всего полносвязанный слой имеет 64 × 6400 = 409 600 весов. И, конечно же, каждый нейрон имеет смещение, соответственно, общее число параметров в слое получается равным 409 664.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 64)           320000
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                409664
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 729,729
Trainable params: 729,729
Non-trainable params: 0
_________________________________________________________________
```

Рис. 11.16. Сводная информация о полносвязанной модели классификации эмоциональной окраски

- Наконец, единственный нейрон в выходном слое имеет 64 веса — по одному для активации выхода каждого нейрона в предыдущем слое — плюс смещение, то есть всего 65 параметров.
- Сложив параметры всех слоев, получаем общее количество, равное 730 000.

Как показано в листинге 11.20, наш полносвязанный классификатор эмоциональной окраски компилируется уже знакомой по предыдущим главам строкой кода, но из-за того что в бинарном классификаторе имеется только один выходной нейрон, вместо функции стоимости `categorical_crossentropy`, использовавшейся в мультиклассовых классификаторах MNIST, здесь применяется функция `binary_crossentropy`.

Листинг 11.20. Компиляция классификатора эмоциональной окраски

```
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

В листинге 11.21 мы создаем объект `ModelCheckpoint()`, который поможет сохранять параметры модели после каждой эпохи обучения. Благодаря этому потом мы сможем вернуться к параметрам, полученным в выбранную эпоху, для оценки модели или прогнозирования в процессе эксплуатации. Если каталог `output_dir` еще не существует, мы вызываем `makedirs()`, чтобы создать его.

Листинг 11.21. Создание объекта и каталога для сохранения параметров модели после каждой эпохи обучения

```
modelcheckpoint = ModelCheckpoint(filepath=output_dir+
                                  "/weights.{epoch:02d}.hdf5")
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/4
25000/25000 [==============================] - 2s 80us/step - loss: 0.5612 - acc: 0.6892 - val_loss: 0.3630 - val_acc: 0.8398
Epoch 2/4
25000/25000 [==============================] - 2s 69us/step - loss: 0.2851 - acc: 0.8841 - val_loss: 0.3486 - val_acc: 0.8447
Epoch 3/4
25000/25000 [==============================] - 2s 70us/step - loss: 0.1158 - acc: 0.9646 - val_loss: 0.4252 - val_acc: 0.8337
Epoch 4/4
25000/25000 [==============================] - 2s 70us/step - loss: 0.0237 - acc: 0.9961 - val_loss: 0.5304 - val_acc: 0.8340
```

Рис. 11.17. Обучение полносвязанного классификатора эмоциональной окраски

Этап обучения модели классификатора эмоциональной окраски (листинг 11.22) должен показаться вам знакомым, как и этап компиляции, за исключением разве что аргумента `callbacks`, который мы использовали для передачи объекта `modelcheckpoint`[1].

Листинг 11.22. Обучение классификатора эмоциональной окраски

```
model.fit(x_train, y_train,
          batch_size=batch_size, epochs=epochs, verbose=1,
          validation_data=(x_valid, y_valid),
          callbacks=[modelcheckpoint])
```

Как показано на рис. 11.17, наименьшие потери (0.349) и наивысшая точность (84.5%) при проверке были достигнуты после второй эпохи. После третьей и четвертой эпох модель оказалась сильно переобученной, о чем говорит намного более высокая точность на обучающем наборе, чем на проверочном. После четвертой эпохи точность на обучающих данных составила 99.6%, а на проверочных данных — намного ниже, всего 83.4%.

Для более тщательной оценки лучшей эпохи мы использовали метод `load_weights()` из библиотеки Keras, с помощью которого загрузили параметры из второй эпохи (`weights.02.hdf5`) обратно в модель, как показано в листинге 11.23[2,3].

---

[1] Это не первый случай использования `callbacks`. Мы уже использовали этот аргумент, в котором можно передать список обратных вызовов, для наблюдения за ходом обучения модели в TensorBoard (см. главу 9).

[2] Несмотря на свое название, `load_weights()` загружает *все* параметры модели, включая смещения. Поскольку веса, как правило, составляют подавляющее большинство параметров, специалисты по глубокому обучению часто называют файлы с параметрами файлами «весов».

[3] В ранних версиях Keras эпохи нумеровались начиная с нуля, но потом они начали нумероваться с 1.

Листинг 11.23. Загрузка параметров модели

```
model.load_weights(output_dir+"/weights.02.hdf5")
```

После выбора лучшей эпохи мы можем найти прогнозные оценки $\hat{y}$ для всех проверочных данных, передав методу `predict_proba()` набор данных `x_valid`, как показано в листинге 11.24.

Листинг 11.24. Прогнозные оценки ŷ для всех проверочных данных

```
y_hat = model.predict_proba(x_valid)
```

Например, обратившись к оценке `y_hat[0]`, можно увидеть оценку эмоциональной окраски первого отзыва в проверочном наборе, предсказанную моделью. Для этого отзыва оценка $\hat{y} = 0.09$, то есть модель оценивает вероятность, что отзыв положительный, как равную 9%, а вероятность, что он отрицательный — 91%. Обратившись к элементу `y_valid[0]`, можно убедиться, что для этого отзыва $\hat{y} = 0$, то есть это действительно отрицательный отзыв, значит, оценка $\hat{y}$ модели достаточно точная! Если вам интересно увидеть содержимое отрицательного отзыва, выполните код из листинга 11.17, изменив его так, чтобы получить полный текст из элемента списка `all_x_valid[0]`, как показано в листинге 11.25.

Листинг 11.25. Вывод полного отзыва из проверочного набора

```
' '.join(index_word[id] for id in all_x_valid[0])
```

Изучение отдельных баллов может представлять определенный интерес, однако общую эффективность модели лучше оценивать по всем результатам проверки вместе. Для этого можно построить гистограмму всех значений $\hat{y}$, полученных на проверочных данных, если запустить код в листинге 11.26.

Листинг 11.26. Создание гистограммы значений ŷ, полученных на проверочных данных

```
plt.hist(y_hat)
_ = plt.axvline(x=0.5, color='orange')
```

Полученная гистограмма представлена на рис. 11.18. Как можно заметить, в большинстве случаев модель четко определяет эмоциональную окраску отзывов: примерно 8000 из 25 000 (~32%) получили $\hat{y}$ меньше 0.1 и ~6500 (~26%) получили $\hat{y}$ больше 0.9.

Вертикальная оранжевая линия на рис. 11.18 отмечает порог 0.5. При использовании упрощенных вычислений отзывы с оценкой $\hat{y}$ выше этого порога можно было бы считать положительными. Но, как обсуждалось выше в этой главе, такой упрощенный подход к классификации может приводить к ошибочным результатам, потому что нельзя утверждать, что эмоциональная окраска отзыва с оценкой $\hat{y}$ чуть меньше 0.5 существенно отличается от окраски отзыва с оценкой $\hat{y}$ чуть больше 0.5. Чтобы получить более детальный анализ эффективности модели как бинарного классификатора, можно использовать метод

`roc_auc_score()` из библиотеки *scikit-learn* для непосредственного вычисления метрики ROC AUC, как показано в листинге 11.27.

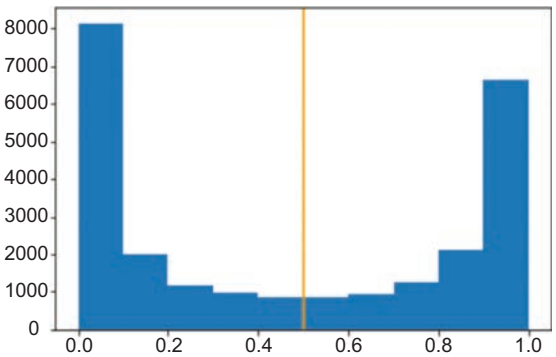


Рис. 11.18. Гистограмма с оценками ŷ, полученными на проверочных данных после второй эпохи обучения полносвязанного классификатора эмоциональной окраски

Листинг 11.27. Вычисление метрики ROC AUC на проверочных данных

```
pct_auc = roc_auc_score(y_valid, y_hat)*100.0
"{:0.2f}".format(pct_auc)
```

Использовав метод `format()` для вывода результатов в удобочитаемом формате, можно увидеть, что площадь под кривой рабочей характеристики приемника составляет 92.9% (довольно много).

Чтобы понять, где модель ошибается, можно создать `DataFrame` на основе множеств значений y и $\hat{y}$ для проверочного набора данных, используя код из листинга 11.28.

Листинг 11.28. Создание объекта DataFrame на основе множеств значений y и $\hat{y}$

```
float_y_hat = []
for y in y_hat:
    float_y_hat.append(y[0])
ydf = pd.DataFrame(list(zip(float_y_hat, y_valid)),
                   columns=['y_hat', 'y'])
```

На рис. 11.19 показаны первые 10 записей в экземпляре `DataFrame`, полученные вызовом `ydf.head(10)`.

Обращаясь к экземпляру `DataFrame`, как показано в листингах 11.29 и 11.30, а затем после изучения отдельных результатов этих запросов меняя индекс списка в листинге 11.25, можно получить более полное представление о том, в каких отзывах модель ошибается больше всего.

Листинг 11.29. Десять отрицательных отзывов в проверочном наборе, получивших высокую оценку ŷ

```
ydf[(ydf.y == 0) & (ydf.y_hat > 0.9)].head(10)
```

Листинг 11.30. Десять положительных отзывов в проверочном наборе, получивших низкую оценку ŷ

```
ydf[(ydf.y == 0) & (ydf.y_hat > 0.9)].head(10)
```

	y_hat	y
0	0.089684	0
1	0.982754	1
2	0.746905	1
3	0.543328	0
4	0.997054	1
5	0.833994	1
6	0.766254	1
7	0.008032	0
8	0.812743	0
9	0.729463	1

Рис. 11.19. Экземпляр DataFrame с метками y и оценками ŷ для проверочного набора данных IMDb

```
"START wow another kevin costner hero movie postman tin cup waterworld bo
dyguard wyatt earp robin hood even that baseball movie seems like he make
s movies specifically to be the center of attention the characters are al
most always the same the heroics the flaws the greatness the fall the red
emption yup within the 1st 5 minutes of the movie we're all supposed to b
e in awe of his character and it builds up more and more from there br br
and this time the story story is just a collage of different movies you d
on't need a spoiler you've seen this movie several times though it had di
fferent titles you'll know what will happen way before it happens this is
like mixing an officer and a gentleman with but both are easily better mo
vies watch to see how this kind of movie should be made and also to see h
ow an good but slightly underrated actor russell plays the hero"
```

Рис. 11.20. Пример ложно-положительного результата: этот отрицательный отзыв был неправильно классифицирован моделью как положительный

На рис. 11.29 представлен пример ложно-положительного результата — отрицательного отзыва ($y = 0$) с очень высокой оценкой ($\hat{y} = 0.97$), — выявленного кодом из листинга 11.29[1]. А на рис. 11.30 показан пример ложного-отрицательного результата — положительного отзыва ($y = 1$) с очень низкой оценкой ($\hat{y} = 0.06$), — выявленного кодом из листинга 11.30[2]. Подобный анализ результатов модели вскрывает один потенциальный недостаток: наш полносвязанный

[1] Мы обнаружили этот конкретный обзор — 387-й в проверочном наборе данных, — выполнив инструкцию: `' '.join(index_word[id] for id in all_x_valid[386])`.

[2] Вы сможете отыскать его, выполнив инструкцию: `' '.join(index_word[id] for id in all_x_valid[224])`.

классификатор не обнаруживает шаблоны, состоящие из нескольких лексем, следующих друг за другом, которые могут определять эмоциональную окраску отзыва к фильму. Например, если бы модель учитывала такие шаблоны, как пара лексем *not-good* (не хороший), она могла бы использовать их как признак отрицательной эмоциональной окраски.

СВЕРТОЧНЫЕ СЕТИ

Как рассказывалось в главе 10, сверточные сети особенно хорошо подходят для обнаружения пространственных шаблонов. В этом разделе мы используем эту их особенность, чтобы выявить пространственные закономерности в последовательности слов — например, *not-good*, — и посмотрим, смогут ли они превзойти нашу полносвязанную сеть в задаче классификации эмоциональной окраски отзывов к фильмам. Весь программный код, создающий и обучающий сверточную сеть, вы найдете в нашем блокноте *convolutional_sentiment_classifier.ipynb*.

```
"START finally a true horror movie this is the first time in years that i
had to cover my eyes i am a horror buff and i recommend this movie but it
is quite gory i am not a big wrestling fan but kane really pulled the whc
le monster thing off i have to admit that i didn't want to see this movie
my 17 year old dragged me to it but am very glad i did during and after t
he movie i was looking over my shoulder i have to agree with others about
the whole remake horror movies enough is enough i think that is why this
movie is getting some good reviews it is a refreshing change and takes yc
u back to the texas chainsaw first one michael myers and jason and no cgi
crap"
```

Рис. 11.21. Пример ложно-отрицательного результата: этот положительный отзыв был неправильно классифицирован моделью как отрицательный

Эта модель имеет те же зависимости, что и предыдущая модель полносвязанного классификатора (см. листинг 11.12), за исключением трех новых типов слоев, как показано в листинге 11.31.

Листинг 11.31. Дополнительные зависимости для сверточной сети

```
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras.layers import SpatialDropout1D
```

В листинге 11.32 представлены гиперпараметры сверточного классификатора эмоциональной окраски.

Листинг 11.32. Гиперпараметры сверточного классификатора эмоциональной окраски

```
# имя каталога для сохранения результатов:
output_dir = 'model_output/conv'

# обучение:
```

```
epochs = 4
batch_size = 128

# векторное пространство слов:
n_dim = 64
n_unique_words = 5000
max_review_length = 400
pad_type = trunc_type = 'pre'
drop_embed = 0.2 # новое!

# архитектура сверточного слоя:
n_conv = 256 # фильтры, они же ядра
k_conv = 3 # длина ядра

# архитектура полносвязанного слоя:
n_dense = 256
dropout = 0.2
```

Сравним их с гиперпараметрами полносвязанного классификатора (см. листинг 11.13):

- Мы создали новый каталог (`'conv'`) для сохранения параметров модели после каждой эпохи обучения.
- Число эпох и размер пакета остались прежними.
- Размерность векторного пространства слов осталась прежней, но:
  - мы в четыре раза увеличили `max_review_length` — до 400, потому что, несмотря на значительное увеличение объема ввода, а также числа скрытых слоев, сверточный классификатор будет иметь намного меньше параметров по сравнению с полносвязанным классификатором;
  - добавили гиперпараметр `drop_embed`, управляющий прореживанием для слоя `Embedding`.
- Сверточный классификатор будет иметь два скрытых слоя, следующих за `Embedding`, создающим векторное пространство:
  - сверточный слой с 256 фильтрами (`n_conv`), каждый с одним измерением и *длиной* фильтра 3 (`k_conv`). Работая с двумерными изображениями в главе 10, мы использовали сверточные слои с двумерными фильтрами. Данные на естественном языке — в письменном или устном виде — имеют только одно измерение (измерение времени), поэтому сверточные слои в этой главе будут иметь одномерные фильтры;
  - полносвязанный слой с 256 нейронами (`n_dense`) и прореживанием на уровне 20%.

Операции, выполняющие загрузку данных IMDb и стандартизацию длин отзывов, идентичны тем, что использовались в блокноте *dense_sentiment_classifier*.

ipynb (см. листинги 11.14 и 11.18). Архитектура модели, конечно, немного отличается и показана в листинге 11.33.

Листинг 11.33. Архитектура сверточного классификатора эмоциональной окраски

```
model = Sequential()

# Векторное пространство слов:
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))

# сверточный слой:
model.add(Conv1D(n_conv, k_conv, activation='relu'))
# model.add(Conv1D(n_conv, k_conv, activation='relu'))
model.add(GlobalMaxPooling1D())

# полносвязанный слой:
model.add(Dense(n_dense, activation='relu'))
model.add(Dropout(dropout))

# выходной слой:
model.add(Dense(1, activation='sigmoid'))
```

Рассмотрим подробнее архитектуру модели:

- Слой `Embedding` тот же, что и прежде, только на этот раз к нему применяется прореживание.
- Мы больше не используем `Flatten()`, потому что слой `Conv1D()` принимает оба измерения в выходе слоя `Embedding`.
- В одномерном сверточном слое мы используем активацию `relu`. Также слой имеет 256 уникальных фильтров, каждый из которых может специализироваться на активациях при прохождении определенной последовательности из трех лексем. Карта активаций для каждого из 256 фильтров имеет длину 398 для выхода с формой 256×398[1].
- Если хотите, можете добавить дополнительные сверточные слои, например, раскомментировав вторую строку с вызовом `Conv1D()`.
- *Объединение по глобальному максимуму* — широко используемый прием уменьшения размерности в моделях NLP глубокого обучения. Мы исполь-

[1] Как рассказывалось в главе 10, когда двумерный фильтр выполняет свертку изображения, мы теряем пикселы по его периметру, если предварительно не добавить в изображение пустые пикселы. В модели естественного языка одномерный сверточный фильтр имеет длину 3, поэтому в самом левом углу отзыва он центрируется по второй лексеме, а в крайнем правом углу — по второй с конца. Поскольку мы не дополняли отзывы с концов пустыми лексемами перед передачей их в сверточный слой, мы теряем часть информации с каждого конца: $400 - 1 - 1 = 398$. Но не будем переживать из-за этой потери.

зуем его здесь для сжатия карты активаций с 256 × 398 до 256 × 1. При применении операции вычисления максимального значения сохраняется только порядок наибольшей активации для данного сверточного фильтра и теряется информация, зависящая от положения на оси времени, которую фильтр может выводить в свою карту активаций длиной 398 элементов.

- Поскольку активации на выходе слоя субдискретизации с объединением по максимальному значению являются одномерными, их можно прямиком передать в полносвязанный слой, который (опять же) состоит из нейронов с функцией активации `relu` и к которому применяется прореживание.
- Выходной слой остается прежним.
- В общей сложности модель имеет 435 000 параметров (рис. 11.22), что на несколько сотен тысяч меньше, чем в полносвязанном классификаторе. Тем не менее каждая эпоха обучения этой модели будет длиться дольше, потому что операция свертки относительно дорогостоящая с вычислительной точки зрения.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 400, 64)           320000
_________________________________________________________________
spatial_dropout1d_1 (Spatial (None, 400, 64)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 398, 256)          49408
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 435,457
Trainable params: 435,457
Non-trainable params: 0
```

Рис. 11.22. Сводная информация о сверточной модели классификатора эмоциональной окраски

Важно отметить, что сверточные фильтры в этой модели обнаруживают тройки не просто *слов*, а *векторов слов*. Согласно обсуждению в главе 2, в отличие от представления в прямом дискретном коде, векторное представление мягко размазывает смысл по многомерному пространству (см. табл. 2.1), поэтому все модели в этой главе специализируются на выявлении *смысловых* связей между словами, а не на простом сопоставлении отдельных слов с эмоциональной окраской. Например, если сеть выяснит, что пара лексем *not-good* служит признаком отрицательного отзыва, то она должна понять и то, что пара *not-great*

(ничего особенного) тоже имеет отрицательную окраску, потому что лексемы *good* и *great* имеют сходные значения (и, следовательно, должны находиться поблизости в векторном пространстве).

Компиляция, сохранение промежуточных результатов и обучение модели осуществляются точно так же, как в примере с полносвязанным классификатором (см. листинги 11.20, 11.21 и 11.22 соответственно). Ход обучения показан на рис. 11.23. Наименьшие потери (0.258) и наивысшая точность (89.6%) на проверочных данных были достигнуты в третью эпоху. Загрузив параметры модели, полученные после этой эпохи (использовав код из листинга 11.23, но указав `weights.03.hdf5`), можно вычислить оценки $\hat{y}$ для всех отзывов в проверочном наборе (точно так же, как в листинге 11.24). Создав гистограмму (рис. 11.24) распределения оценок $\hat{y}$ (использовав код из листинга 11.26), можно наглядно увидеть, что сверточная сеть намного более однозначно определяет эмоциональную окраску, чем полносвязанная сеть (см. рис. 11.18): еще около тысячи отзывов получили оценку $\hat{y} < 0.1$ и несколько тысяч — оценку $\hat{y} > 0.9$. Вычислив метрику ROC AUC (использовав код из листинга 11.27), мы получаем очень высокий балл — 96.12%, указывающий, что наши надежды на успех сверточной сети оправдались в полной мере: это заметное улучшение по сравнению с высоким (~93%) баллом полносвязанной сети.

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/4
25000/25000 [==============================] - 41s 2ms/step - loss: 0.4894 - acc: 0.7447 - val_loss: 0.2971 - val_acc: 0.8750
Epoch 2/4
25000/25000 [==============================] - 41s 2ms/step - loss: 0.2534 - acc: 0.8972 - val_loss: 0.2604 - val_acc: 0.8914
Epoch 3/4
25000/25000 [==============================] - 41s 2ms/step - loss: 0.1709 - acc: 0.9357 - val_loss: 0.2577 - val_acc: 0.8959
Epoch 4/4
25000/25000 [==============================] - 41s 2ms/step - loss: 0.1151 - acc: 0.9589 - val_loss: 0.2828 - val_acc: 0.8934
```

Рис. 11.23. Обучение сверточного классификатора эмоциональной окраски

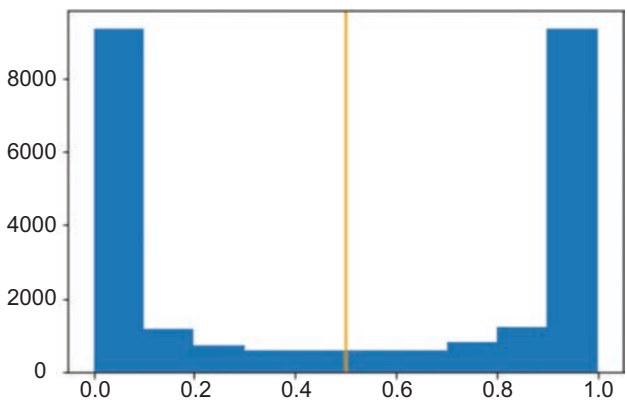


Рис. 11.24. Гистограмма с оценками ŷ, полученными на проверочных данных после третьей эпохи обучения сверточного классификатора эмоциональной окраски

СЕТИ, СПЕЦИАЛИЗИРУЮЩИЕСЯ НА ИЗУЧЕНИИ ПОСЛЕДОВАТЕЛЬНЫХ ДАННЫХ

Наш сверточный классификатор превзошел полносвязанную сеть — возможно, благодаря тому что сверточный слой лучше справляется с выявлением шаблонов, предсказывающих некоторый результат, например положительной или отрицательной эмоциональной окраски отзывов. Фильтры в сверточных слоях отлично подходят для определения коротких последовательностей, таких как тройки слов (напомним, что в листинге 11.32 мы установили `k = 3`), но документ на естественном языке, такой как отзыв к фильму, может содержать намного более длинные последовательности слов, которые, если рассматривать все вместе, позволили бы модели точнее предсказать результат. Для обработки длинных последовательностей данных существует семейство моделей глубокого обучения, называемых рекуррентными нейронными сетями (Recurrent Neural Network, RNN), которые включают специализированные типы слоев — *ячейки долгой краткосрочной памяти* (Long Short-Term Memory, LSTM) и *управляемые рекуррентные блоки* (Gated Recurrent Unit, GRU). В этом разделе мы познакомимся с теоретическими основами RNN и используем некоторые виды рекуррентных сетей для решения нашей задачи классификации отзывов к фильмам. Мы также познакомимся с понятием *внимания* (attention) — более сложным подходом к моделированию данных на естественном языке, который устанавливает новые критерии для приложений NLP.

Как упоминалось в начале главы, семейство сетей RNN, включая LSTM и GRU, хорошо подходит для обработки не только данных на естественном языке, но и любых других последовательных данных. К их числу относятся данные о ценах (например, финансовые показатели, цены на акции), объемах продаж, температуре и показателях заболеваемости (эпидемиология). Применение RNN кроме как для задач NLP выходит за рамки этой книги, поэтому, если вам интересна тема моделирования количественных данных во времени, обращайтесь к разделу *Time Series Prediction* на сайте *jonkrohn.com/resources*.

РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ

Рассмотрим следующие предложения:

Jon and Grant are writing a book together. They have really enjoyed writing it[1].

Человеческий разум легко определит смысл местоимений во втором предложении. Вы без труда поймете, что местоимение «they» (им) во втором предложении относится к авторам, а «it» (ее) — к книге, которую они написали. Это легкая задача для вас, но очень сложная для нейронной сети.

[1] Джон и Грант вместе написали книгу. Им очень понравилось писать ее. — *Примеч. пер.*

Сверточный классификатор эмоциональной окраски, который мы создали в предыдущем разделе, мог рассматривать каждое слово в контексте всего лишь двух соседних с каждой стороны от него (`k_conv = 3`, как показано в листинге 11.32). При таком небольшом окне эта нейронная сеть не имела возможности понять, на что могут ссылаться местоимения «they» или «it». Человеческий мозг способен на это, потому что наши мысли вращаются вокруг друг друга, и мы постоянно возвращаемся к более ранним из них, чтобы понять текущий контекст. В этом разделе мы познакомим вас с идеей рекуррентных нейронных сетей, предназначенных именно для этого: в их структуру встроены циклы, которые обеспечивают сохранность информации с течением времени.

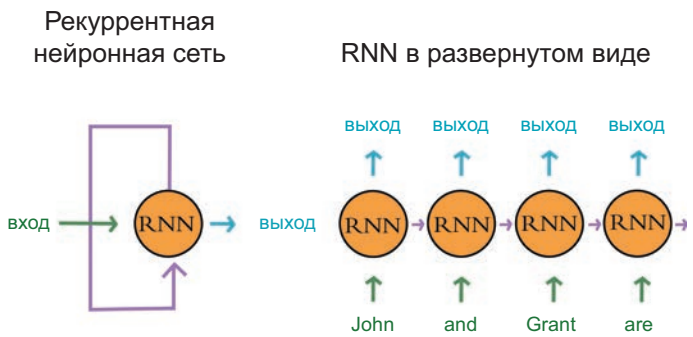


Рис. 11.25. Схематическая диаграмма рекуррентной нейронной сети

На рис. 11.25 показана обобщенная структура рекуррентной нейронной сети (RNN). Фиолетовая линия слева обозначает петлю, по которой информация передается между шагами в сети. Для каждого входа, так же как в полносвязанной сети, имеется свой нейрон. Это видно на рис. 11.25 справа, где изображена структура RNN в развернутом виде. Каждому слову в предложении соответствует свой рекуррентный модуль (для краткости здесь показаны только первые четыре слова)[1]. Однако каждый модуль получает дополнительный вход от предыдущего, благодаря чему сеть может передавать информацию из более ранних шагов в последовательности. В случае, показанном на рис. 11.25, каждое слово представлено как отдельный временной шаг, поэтому сеть способна понять, что книгу писали «Jon» и «Grant», и связать эти понятия со словом «they», следующим далее в этой последовательности.

Рекуррентные нейронные сети имеют более высокую вычислительную сложность на этапе обучения, чем сети прямого распространения, такие как полно-

[1] Это еще одна причина, почему необходимо дополнять короткие предложения на этапе предварительной обработки: RNN ожидает последовательность определенной длины, поэтому, если она слишком короткая, мы добавляем лексемы `PAD`, чтобы компенсировать разницу.

связанные и сверточные сети, которые мы использовали до этого. Как показано на рис. 8.6, сети прямого распространения включают этап обратного распространения стоимости в направлении от выходного слоя к входному. Если сеть включает рекуррентный слой (например, `SimpleRNN`, `LSTM` или `GRU`), тогда стоимость должна распространяться в обратном направлении не только от выходного слоя к входному, но и обратно по временным шагам рекуррентного слоя (от более поздних временных шагов к более ранним). Обратите внимание, что точно так же, как затухает градиент при обратном распространении от более поздних скрытых слоев к более ранним (см. рис. 8.8), он затухает и при обратном распространении от более поздних временных шагов в рекуррентном слое к более ранним. Из-за этого более поздние временные шаги сильнее влияют на результат, чем более ранние[1].

РЕАЛИЗАЦИЯ RNN С ПОМОЩЬЮ KERAS

Библиотека Keras позволяет легко добавить рекуррентный слой в архитектуру нейронной сети и получить RNN, как показано в нашем блокноте Jupyter *rnn_sentiment_classifier.ipynb*. Обратите внимание, что для краткости и удобства следующие ячейки с кодом идентичны во всех блокнотах Jupyter, упоминаемых в этой главе, включая блокноты *dense_sentiment_classifier.ipynb* и *convolutional_sentiment_classifier.ipynb*, которые мы уже рассмотрели:

- Загрузка зависимостей (листинг 11.12), за исключением того, что часто в разных блокнотах могут иметься одна или две дополнительные зависимости. Мы отмечаем эти различия отдельно — как правило, при представлении архитектуры нейронной сети.
- Загрузка корпуса отзывов к фильмам из IMDb (листинг 11.14).
- Стандартизация отзывов по длине (листинг 11.18).
- Компиляция модели (листинг 11.20).
- Создание объекта `ModelCheckpoint()` и каталога (листинг 11.21).
- Обучение модели (листинг 11.22).
- Загрузка параметров модели из наилучшей эпохи (листинг 11.23) с одним важным исключением: выбор конкретной эпохи для загрузки зависит от того, какая из них имеет наименьшие потери на проверочных данных.
- Вычисление оценок $\hat{y}$ для всех проверочных данных (листинг 11.24).

---

[1] Если вы чувствуете, что начало последовательности (например, слова в начале отзыва к фильму) в большей степени относится к решаемой задаче (классификация эмоциональной окраски), чем конец (слова в конце отзыва), вы можете изменить это перед передачей в сеть, чтобы обеспечить надежное обратное распространение начала последовательности в рекуррентных слоях до конца.

- Построение гистограммы распределения оценок $\hat{y}$ (листинг 11.26).
- Вычисление метрики ROC AUC (листинг 11.27).

Отличаются только следующие ячейки с кодом:

1. Устанавливающие гиперпараметры.
2. Определяющие архитектуру нейронной сети.

Настройка гиперпараметров для нашей RNN показана в листинге 11.34.

Листинг 11.34. Гиперпараметры рекуррентной нейронной сети классификатора эмоциональной окраски

```
# имя каталога для сохранения результатов:
output_dir = 'model_output/rnn'

# обучение:
epochs = 16 # теперь их намного больше!
batch_size = 128

# векторное пространство слов:
n_dim = 64
n_unique_words = 10000
max_review_length = 100 # уменьшена из-за эффекта затухания градиента
                        # с течением времени
pad_type = trunc_type = 'pre'
drop_embed = 0.2

# архитектура рекуррентного слоя:
n_rnn = 256
drop_rnn = 0.2
```

Вот некоторые отличия этого блокнота от предыдущих:

- Мы увеличили число эпох обучения до 16, потому что рекуррентные сети менее подвержены переобучению.
- Мы вновь уменьшили `max_review_length` до `100`, хотя для простой RNN даже это значение слишком велико. При такой длине с помощью LSTM (описывается в следующем разделе) в обратном направлении может распространяться до 100 временных шагов (то есть до 100 лексем или слов, следующих друг за другом) до полного исчезновения градиента обучения, но в простой сети RNN градиент полностью затухает уже после примерно 10 временных шагов. Поэтому значение `max_review_length`, по всей видимости, можно уменьшить до `10` или даже меньше, прежде чем снижение эффективности этой модели станет заметно.
- Экспериментируя со всеми рекуррентными архитектурами, представленными в этой главе, мы пробовали удваивать размер словаря до `10 000` лексем. Как нам показалось, это способствовало увеличению точности классификации, хотя мы не выполняли строгих проверок.

- Мы установили `n_rnn = 256`, поэтому можно сказать, что этот рекуррентный слой имеет 256 *модулей*, или *ячеек*. Точно так же, как 256 фильтров позволили нашей сверточной модели сосредоточиться на выявлении 256 уникальных троек значений слов[1], этот параметр позволит рекуррентной сети выявить 256 уникальных последовательностей значений слов, которые могут иметь отношение к оценке эмоциональной окраски.

В листинге 11.35 приводится определение архитектуры рекуррентной модели.

Листинг 11.35. Архитектура рекуррентного классификатора эмоциональной окраски

```
from keras.layers import SimpleRNN

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(SimpleRNN(n_rnn, dropout=drop_rnn))
model.add(Dense(1, activation='sigmoid'))
```

Вместо сверточного или полносвязанного слоя (или обоих) в этой модели создается слой вызовом метода `SimpleRNN()`, которому можно передать аргумент `dropout`; благодаря этому отпадает необходимость добавлять прореживание в отдельной строке кода. После рекуррентного слоя, в отличие от сверточного, редко добавляется полносвязанный, потому что это почти не дает никаких преимуществ в эффективности прогнозирования. Однако вы можете попробовать добавить скрытый слой `Dense()` и посмотреть, что из этого выйдет.

Результаты работы этой модели (которые показаны в нашем блокноте *rnn_sentiment_classifier.ipynb*) не внушают оптимизма. Мы обнаружили, что потери постепенно снижались в течение первой полудюжины эпох, а потом начали прыгать. Это признак того, что модель столкнулась со сложностями выявления шаблонов даже внутри обучающих данных, не говоря о проверочных, что случается крайне редко. Фактически все модели, которые мы обучали до сих пор, показывали последовательное снижение потерь на обучающих данных с каждой эпохой.

Потери на проверочных данных тоже начали скакать. Самые низкие показатели (0.504) наблюдались в седьмой эпохе, что соответствовало точности 77.6% на проверочных данных и метрике ROC AUC 84.9%. Все три параметра имеют худшие значения среди всех моделей классификаторов эмоциональной окраски. Это связано с тем, что, как упоминалось выше, обратное распространение в сети RNN действенно только на расстояниях до ~10 временных шагов, после чего градиент уменьшится настолько, что приращения параметров станут

[1] Под «значениями слов» здесь понимается их местоположение в векторном пространстве.

пренебрежимо малыми. Из-за этого простые рекуррентные сети (RNN) редко используются на практике — гораздо чаще встречаются более сложные типы рекуррентных уровней, такие как LSTM, в которых обратное распространение сохраняет действенность на расстояниях до ~100 временных шагов[1].

ДОЛГАЯ КРАТКОСРОЧНАЯ ПАМЯТЬ

Как отмечалось в конце предыдущего раздела, простые архитектуры RNN можно использовать, только если расстояние между информацией и контекстом, в котором она необходима, невелико (менее 10 временных шагов). Однако если для решения задачи требуется учитывать более широкий контекст (что часто имеет место в задачах NLP), предпочтительнее использовать рекуррентные слои другого типа: ячейки долгой краткосрочной памяти (Long Short-Term Memory, LSTM).

Разновидность LSTM рекуррентных архитектур была предложена Зеппом Хохрайтером (Sepp Hochreiter) и Юргеном Шмидхубером (Jürgen Schmidhuber) в 1997 году[2] и в настоящее время наиболее широко используется в приложениях глубокого обучения для обработки естественного языка. Базовая структура слоя LSTM похожа на структуру простых рекуррентных слоев, показанную на рис. 11.25. Слои LSTM точно так же принимают последовательности входных данных (например, лексемы из документа на естественном языке), а также входные данные из предыдущего момента времени. Разница лишь в том, что в каждой ячейке в простом рекуррентном уровне (например, `SimpleRNN()`) имеется единственная функция активации, такая как `tanh`, которая преобразует входные данные ячейки RNN в выходные. Ячейки в слое LSTM, напротив, имеют гораздо более сложную структуру, как показано на рис. 11.26.

Эта схема может показаться пугающе сложной, и мы готовы признать, что такое детальное представление каждого компонента внутри ячейки LSTM избыточно для задач этой книги[3]. Тем не менее есть несколько ключевых моментов, которые мы должны отметить. Первый — *состояние ячейки*, пересекающее LSTM в верхней ее части на рис. 11.26. Обратите внимание, что состояние ячейки не проходит через какие-либо нелинейные функции активации — оно претерпевает лишь незначительные линейные преобразования и просто передается от

---

[1] Единственная ситуация, когда простая сеть RNN может иметь практическое значение, — если последовательности, имеющие отношение к задаче, которую вы решаете с помощью такой модели, содержат не более 10 временных шагов. Это может иметь место в некоторых задачах прогнозирования временных рядов или если в вашем наборе данных имеются только очень короткие строки на естественном языке.

[2] *Hochreiter S. & Schmidhuber J.* (1997). «Long short-term memory». *Neural Computation, 9*, 1735–80.

[3] За подробным описанием ячеек LSTM мы рекомендуем обратиться к иллюстрированной статье Кристофера Олаха (Christopher Olah), доступной по адресу bit.ly/colahLSTM.

ячейки к ячейке. Эти два линейных преобразования (умножение и сложение) являются точками, в которых ячейка в слое LSTM может добавить информацию в состояние для передачи в следующую. В любом случае *перед* добавлением информации в состояние ячейки выполняется сигмоидная активация (обозначена на рисунке буквой σ). Поскольку сигмоидная активация дает в результате значение от 0 до 1, она действует как вентиль, который определяет, будет ли новая информация (из текущего временного шага) добавлена в состояние ячейки или нет.

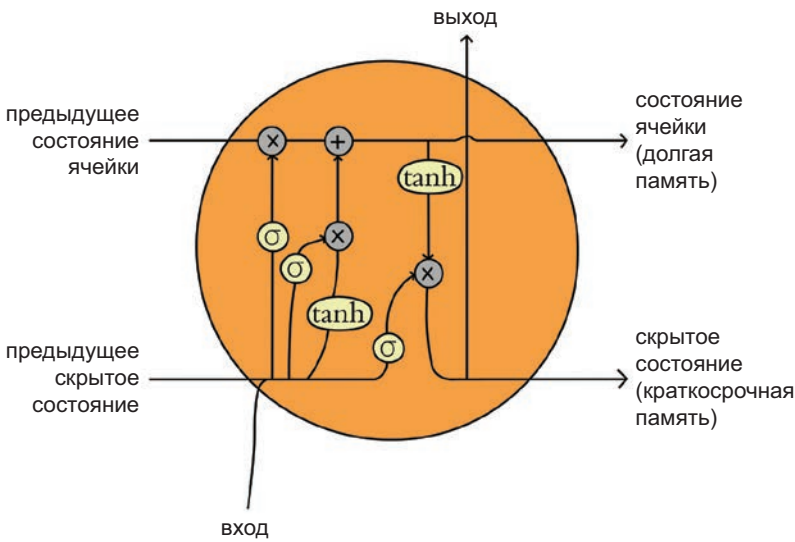


Рис. 11.26. Схематическая диаграмма LSTM

Новая информация на текущем временном шаге — это простое объединение входных данных для текущего временного шага и скрытого состояния из предыдущего временного шага. Это объединение может быть включено в состояние ячейки линейно или после нелинейной активации, но как бы то ни было, именно сигмоидный вентиль принимает решение об объединении информации.

Когда LSTM определит, какую информацию добавить в состояние ячейки, другой сигмоидный вентиль решит, будет ли информация из текущего **входа** добавлена в конечное состояние ячейки. В результате получится **выход** для текущего временного шага. Обратите внимание, что тот же **выход**, но под другим названием («скрытое состояние») передается следующей ячейке LSTM (представляющей следующий временной шаг в последовательности), где объединяется со **входом** следующего временного шага. После этого процесс повторяется, и конечное состояние ячейки (наряду со скрытым состоянием) передается в ячейку, представляющую следующий временной шаг.

Мы понимаем, что это было довольно сложное объяснение. Вот как еще можно описать суть LSTM:

- Состояние ячейки позволяет информации сохраняться по всей длине последовательности, в каждом временном шаге в данной ячейке LSTM. Это — долгая память LSTM.
- Скрытое состояние аналогично рекуррентным соединениям в простой рекуррентной сети и играет роль краткосрочной памяти LSTM.
- Каждая ячейка представляет конкретную точку в последовательности данных (например, конкретную лексему в документе на естественном языке).
- На каждом временном шаге принимается несколько решений (с использованием сигмоидных вентилей) о соответствии информации на этом конкретном временном шаге локальному (скрытое состояние) и глобальному (состояние ячейки) контекстам.
- Первые два вентиля определяют, относится ли информация из текущего временного шага к глобальному контексту (состоянию ячейки) и как она будет объединена в этот поток.
- Конечный вентиль определяет, относится ли информация из текущего временного шага к локальному контексту (необходимость ее добавления в скрытое состояние, которое раздваивается и подается на *выход* текущего временного шага).

Мы советуем уделить немного времени и посмотреть, сможете ли вы, глядя на рис. 11.26, рассказать, как информация движется через ячейку LSTM. Вам будет проще, если вы вспомните, что решение о передаче информации дальше принимают сигмоидные вентили. В любом случае, вот основные выводы из этого раздела:

- Простые рекуррентные ячейки передают между временными шагами информацию только одного типа (скрытое состояние) и содержат только одну функцию активации.
- Ячейки LSTM заметно сложнее: они передают между временными шагами информацию двух типов (скрытое состояние и состояние ячейки) и содержат пять функций активации.

РЕАЛИЗАЦИЯ LSTM С ПОМОЩЬЮ KERAS

Несмотря на дополнительную вычислительную сложность, сети со слоями LSTM реализуются просто, как демонстрирует наш блокнот *lstm_sentiment_classifier.ipynb*. Как можно видеть в листинге 11.36, для нашей сети LSTM мы выбрали те же гиперпараметры, что и для простой RNN, за исключением следующих:

- Для каталога с результатами выбрано другое имя.
- Изменены имена переменных `n_lstm` и `drop_lstm`.
- Число эпох обучения уменьшено до 4, потому что сети LSTM начинают переобучаться намного раньше, чем простые сети RNN.

Листинг 11.36. Гиперпараметры классификатора LSTM эмоциональной окраски

```
# имя каталога для сохранения результатов:
output_dir = 'model_output/LSTM'

# обучение:
epochs = 4
batch_size = 128

# параметры векторного пространства слов:
n_dim = 64
n_unique_words = 10000
max_review_length = 100
pad_type = trunc_type = 'pre'
drop_embed = 0.2

# архитектура сети LSTM:
n_lstm = 256
drop_lstm = 0.2
```

Модель LSTM имеет ту же структуру, что и предыдущая модель RNN, только слой `SimpleRNN()` был заменен на `LSTM()`; см. листинг 11.37.

Листинг 11.37. Архитектура классификатора LSTM эмоциональной окраски

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(LSTM(n_lstm, dropout=drop_lstm))
model.add(Dense(1, activation='sigmoid'))
```

Результаты обучения LSTM представлены в нашем блокноте *lstm_sentiment_classifier.ipynb*. В завершение отметим, что потери на обучающих данных неуклонно снижались от эпохи к эпохе, а значит, процесс обучения модели LSTM протекает более согласованно, чем в простой модели RNN. Однако результаты не впечатляют. Несмотря на относительную сложность, модель LSTM осталась на уровне базовой полносвязанной модели. Наименьшие потери на проверочных данных (0.349) были получены после второй эпохи; точность на проверочных данных составила 84.8%, а метрика ROC AUC — 92.8%.

ДВУНАПРАВЛЕННЫЕ LSTM

Ячейки с двунаправленной долгой краткосрочной памятью (или Bi-LSTM) — это разновидность ячеек с обычной долгой краткосрочной памятью. Если последние предусматривают обратное распространение только в одном направлении (как правило, назад по временным шагам, например от конца отзыва к началу), то ячейки с двунаправленной долгой краткосрочной памятью реализуют обратное распространение в *обоих* направлениях (назад *и вперед* по временным шагам) через некоторый одномерный вход. Это дополнительное направление обратного распространения удваивает вычислительную сложность, но если для вашего приложения точность имеет первостепенное значение, то такой подход вполне оправдан: ячейки Bi-LSTM часто используются в современных приложениях NLP, потому что они способны выявлять шаблоны до и после данной лексемы во входном документе и повышать эффективность модели.

Мы с легкостью можем преобразовать нашу архитектуру LSTM (листинг 11.37) в архитектуру Bi-LSTM. Для этого достаточно заключить слой `LSTM()` в обертку `Bidirectional()`, как показано в листинге 11.38.

Листинг 11.38. Архитектура классификатора Bi-LSTM эмоциональной окраски

```
from keras.layers import LSTM
from keras.layers.wrappers import Bidirectional # новое!

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(Bidirectional(LSTM(n_lstm, dropout=drop_lstm)))
model.add(Dense(1, activation='sigmoid'))
```

Простое преобразование архитектуры LSTM в Bi-LSTM существенно повысило качество модели, как показывают результаты обучения (полностью представлены в нашем блокноте *bi_lstm_sentiment_classifier.ipynb*). Наименьшие потери на проверочных данных (0.331) были достигнуты после четвертой эпохи, при этом точность на проверочных данных после этой эпохи составила 86.0%, а метрика ROC AUC — 93.5%, что переместило эту модель на второе место после сверточной архитектуры.

МНОГОСЛОЙНЫЕ РЕКУРРЕНТНЫЕ МОДЕЛИ

Объединить несколько рекуррентных слоев (`SimpleRNN()`, `LSTM` или другого типа) *сложнее*, чем несколько полносвязанных или сверточных слоев, но не намного: нужно только указать дополнительный аргумент в определении слоя.

Как мы уже говорили, рекуррентные слои принимают упорядоченные последовательности входных данных. *Рекуррентная* (периодическая) природа этих слоев обусловлена последовательной обработкой каждого временного шага и передачей скрытого состояния на вход следующего. На последнем временном шаге выход рекуррентного слоя является одновременно последним скрытым состоянием.

То есть чтобы объединить несколько рекуррентных слоев, нужно использовать аргумент `return_sequence = True`. Он требует, чтобы рекуррентный уровень возвращал скрытые состояния для всех шагов, выполняемых в слое. В результате вывод такого слоя будет иметь три измерения, соответствующие измерениям его входной последовательности. По умолчанию рекуррентный слой передает следующему только последнее скрытое состояние. Этот подход можно использовать, когда информация передается, скажем, в полносвязанный слой. Но если следующим слоем в сети должен быть другой рекуррентный слой, то он должен получить на входе полноценную последовательность. Поэтому чтобы передать в следующий рекуррентный слой массив скрытых состояний всех временных шагов (вместо единственного конечного скрытого состояния), нужно добавить дополнительный аргумент `return_sequence` со значением `True`[1].

Чтобы увидеть, как действует такая сеть, рассмотрим двухслойную модель Bi-LSTM, представленную в листинге 11.39. (Обратите внимание, что в последнем рекуррентном слое мы оставляем значение по умолчанию `return_sequence = False`, чтобы этот конечный рекуррентный слой возвращал только конечное скрытое состояние.)

Листинг 11.39. Архитектура многослойной рекуррентной модели

```
from keras.layers import LSTM
from keras.layers.wrappers import Bidirectional

model = Sequential()
model.add(Embedding(n_unique_words, n_dim,
                    input_length=max_review_length))
model.add(SpatialDropout1D(drop_embed))
model.add(Bidirectional(LSTM(n_lstm_1, dropout=drop_lstm,
                             return_sequences=True))) # новое!
model.add(Bidirectional(LSTM(n_lstm_2, dropout=drop_lstm)))
model.add(Dense(1, activation='sigmoid'))
```

Как уже не раз наблюдалось выше, после главы 1, дополнительные слои позволяют модели нейронной сети выявлять все более сложные и абстрактные

[1] Существует еще один дополнительный аргумент — `return_state` (который, как и `return_sequence`, по умолчанию имеет значение `False`). Он требует от сети кроме конечного скрытого состояния вернуть также конечное состояние ячейки. Этот аргумент редко используется на практике, но может пригодиться, когда потребуется инициализировать состояние ячейки рекуррентного слоя состоянием другого слоя, как это часто делается в моделях «кодер-декодер» (представленных в следующем разделе).

представления. В данном случае увеличенная способность к абстрагированию за счет дополнительного слоя Bi-LSTM вылилась в прирост точности. Многослойная сеть Bi-LSTM существенно превзошла свою однослойную предшественницу с метрикой ROC AUC 94.9% и точностью на проверочных данных 87.8% в лучшую эпоху (вторую, достигшую уровня потерь на проверочных данных 0.296). Полные результаты представлены в нашем блокноте *stacked_bi_lstm_sentiment_classifier.ipynb*.

Тем не менее точность нашей многослойной архитектуры Bi-LSTM, даже при том что она значительно сложнее нашей сверточной архитектуры и разрабатывалась специально для обработки последовательных данных, таких как тексты на естественном языке, хуже, чем у сверточной модели. Возможно, более тонкая настройка гиперпараметров поможет добиться более высоких результатов, но вообще мы считаем, что наши модели LSTM не смогли продемонстрировать весь свой потенциал из-за небольшого объема набора с отзывами к фильмам. Мы полагаем, что увеличение объема данных на естественном языке будет способствовать эффективному обратному распространению через множество временных шагов в слоях LSTM[1].

В семействе рекуррентных сетей ячейки LSTM имеют родственные им управляемые рекуррентные блоки (Gated Recurrent Unit, GRU)[2]. Блоки GRU имеют чуть меньшую вычислительную сложность, чем LSTM, поскольку включают только три функции активации, но тем не менее в своей эффективности они редко уступают ячейкам LSTM. Если немного большая вычислительная сложность для вас не проблема, выбор GRU не даст вам никаких преимуществ по сравнению с LSTM. Но если вы решите все же попробовать модули GRU в Keras, просто импортируйте тип слоев `GRU()` и вставьте его в архитектуру модели вместо `LSTM()`. Практический пример вы найдете в нашем блокноте *gru_sentiment_classifier.ipynb*.

SEQ2SEQ И МЕХАНИЗМ ВНИМАНИЯ

Методы обработки естественного языка, включающие так называемые модели «*последовательность в последовательность*» (sequence-to-sequence, seq2seq), принимают на входе одну последовательность и генерируют на выходе другую. *Нейронный машинный перевод* (Neural Machine Translation, NMT) — типичный пример использования моделей seq2seq, а примером практического использования NMT может служить алгоритм машинного перевода Google Translate[3].

---

[1] Если вы захотите проверить наше мнение, мы предоставим соответствующие рекомендации в главе 14.

[2] *Cho K.* et al. (2014). «Learning phrase representations using RNN encoder-decoder for statistical machine translation». *arXiv:1406.1078*.

[3] Проект Google Translate начал использовать NMT с 2016 года. Подробнее об этом можно почитать по адресу bit.ly/translateNMT.

Модели нейронного машинного перевода имеют структуру *кодер-декодер*, в которой кодер обрабатывает входную последовательность, а декодер генерирует выходную. Оба, кодер и декодер, являются рекуррентными сетями, поэтому на этапе кодирования имеется скрытое состояние, которое передается между ячейками RNN. В конце этапа кодирования конечное скрытое состояние передается в декодер; это конечное состояние можно назвать контекстом. То есть декодер *начинает* с контекста, описывающего входную последовательность. Хотя эта идея имеет теоретическое обоснование, контекст часто оказывается слабым местом: моделям трудно обрабатывать действительно длинные последовательности, из-за чего контекст теряет свою силу.

Механизм внимания был разработан для ликвидации таких проблем[1]. Проще говоря, вместо одного вектора скрытого состояния (последнего) при использовании механизма внимания из кодера в декодер передается полная последовательность скрытых состояний. Каждое из них связано с одним шагом во входной последовательности, хотя декодеру может потребоваться контекст из нескольких, чтобы скорректировать свое поведение на любом данном шаге во время декодирования. С этой целью на каждом шаге декодер вычисляет оценку для каждого из скрытых состояний кодера. Каждое скрытое состояние кодера умножается на его оценки softmax[2]. Это помогает усилить наиболее релевантные контексты (они будут иметь высокие оценки и, следовательно, более высокие вероятности softmax) и ослабить нерелевантные. По сути, механизм внимания взвешивает контексты, доступные для данного временного шага. Взвешенные скрытые состояния суммируются, и новый вектор контекста используется для прогнозирования выхода на каждом последующем временном шаге декодера. При таком подходе модель выборочно анализирует то, что ей известно о входной последовательности, и использует только релевантную информацию, где это необходимо, для формирования выхода. Она *обращает внимание* на наиболее релевантные элементы для всего предложения!

Если бы эта книга была посвящена исключительно обработке естественного языка, мы могли бы добавить хотя бы одну главу, посвященную seq2seq и механизму внимания. Но, так как у нас нет такой возможности, продолжим исследование этих методов, поднимающих планку эффективности многих приложений NLP.

ПЕРЕНОС ОБУЧЕНИЯ В NLP

Практикам компьютерного зрения в течение ряда лет помогала доступность детализированных моделей, предварительно обученных на больших наборах

---

[1] *Bahdanau D.* et al. (2014). «Neural machine translation by jointly learning to align and translate». *arXiv:1409.0473*.

[2] Как рассказывалось в главе 6, функция softmax принимает вектор действительных чисел и генерирует распределение вероятностей с количеством классов, совпадающим с числом элементов во входном векторе.

данных. Как было описано в разделе «Перенос обучения» в конце главы 10, любой пользователь может загрузить предварительно обученные модели и быстро развить их для конкретных применений в приложениях компьютерного зрения. В последнее время такой перенос обучения стал возможен и для задач NLP[1].

Первым появился прием ULMFiT (*U*niversal *L*anguage *M*odel *Fi*ne-*T*uning — универсальная точная настройка языковой модели). В статье с описанием этого приема были представлены инструменты с открытым исходным кодом, позволяющие другим использовать многое из того, что выявила модель во время предварительного обучения[2]. Соответственно, такие модели можно точно настроить на решение конкретных задач за меньшее время и с меньшим объемом данных и добиться высокой точности.

Вскоре после этого был разработан прием ELMo (*E*mbeddings from *L*anguage *Mo*dels — встраивание векторных представлений из языковых моделей)[3]. В этом усовершенствованном методе на основе векторов слов, с которыми мы познакомились в этой главе, векторные представления зависят не только от самих слов, но и от контекста. Метод ELMo не назначает фиксированные векторные представления всем словам в словаре, а исследует контекст каждого слова, прежде чем назначить ему конкретный вектор. Модель ELMo предварительно обучается на очень большом корпусе; если бы вам самим пришлось обучать такую модель, понадобились бы значительные вычислительные ресурсы, но теперь вы можете использовать ее как составной компонент в своих моделях NLP.

Последний метод переноса обучения, которой мы рассмотрим, называется BERT (*B*i-directional *E*ncoder *R*epresentations from *T*ransformers — двунаправленный кодировщик представлений, полученных от преобразователей) и был разработан в Google[4]. Предварительно обученные модели BERT, настроенные на решение конкретных задач NLP, пожалуй, в еще большей степени, чем ULMFiT и ELMo, способствовали большим достижениям в широком спектре современных приложений и требуют гораздо меньше времени на обучение и меньшего объема данных.

---

[1] В процессе знакомства со слоями `Embedding()` в начале этой главы мы затронули тему переноса обучения с использованием векторов слов. Приемы переноса обучения, описанные в этом разделе — ULMFiT, ELMo и BERT, — по духу близки к задачам компьютерного зрения, потому что (подобно иерархическим визуальным признакам, которые представляют глубокие сверточные сети; см. рис. 1.17) они позволяют создавать иерархические представления элементов естественного языка (таких, как части слов, слова и контекст, как показано на рис. 2.9). Векторы слов, напротив, не имеют иерархии; они охватывают только уровень слов.

[2] *Howard J. and Ruder S.* (2018). «Universal language model fine-tuning for text classification». *arXiv:1801.06146*.

[3] *Peters M. E.* et al. (2018). «Deep contextualized word representations». *arXiv:1802.05365*.

[4] *Devlin J.* et al. (2018). «BERT: Pre-training of deep bidirectional transformers for language understanding». *arXiv: 0810.04805*.

НЕПОСЛЕДОВАТЕЛЬНЫЕ АРХИТЕКТУРЫ: ФУНКЦИОНАЛЬНЫЙ API В БИБЛИОТЕКЕ KERAS

Решить данную задачу можно бесчисленным множеством способов, которые позволяют объединить типы слоев, уже рассмотренные в этой книге, и сформировать архитектуру модели глубокого обучения. Например, загляните в наш блокнот *conv_lstm_stack_sentiment_classifier.ipynb*, где мы использовали творческий подход и разработали модель, включающую сверточный слой, который передает свои активации в слой Bi-LSTM[1]. Однако до сих пор мы были ограничены использованием модели `Sequential()`, для которой необходимо, чтобы каждый слой перетекал непосредственно в следующий.

Несмотря на то что последовательные модели составляют подавляющее большинство моделей глубокого обучения, иногда бывает оправданным использование зачастую более сложных непоследовательных архитектур, допускающих бесконечные возможности в проектировании моделей[2]. В таких ситуациях можно использовать преимущества *функционального API* библиотеки Keras, основанного на использовании класса `Model` вместо `Sequential`, с которым мы работали до сих пор.

В качестве примера реализации непоследовательной архитектуры мы решили взять наш самый эффективный классификатор эмоциональной окраски, сверточную модель, и посмотреть, можно ли выжать больше сока из пресловутого лимона. Как показано на рис. 11.27, идея заключается в том, чтобы создать три параллельных потока сверточных слоев, каждый из которых получает векторы слов из `Embedding()`. Так же как в блокноте *convolutional_sentiment_classifier.ipynb*, один из этих потоков будет иметь длину фильтра в три лексемы. Другой — в *две* лексемы, чтобы он мог специализироваться на выявлении пар векторов слов, которые, как представляется, имеют отношение к классификации отзывов на положительные и отрицательные. Третий сверточный поток будет иметь длину фильтра в *четыре* лексемы и специализироваться на выявлении значений четверок слов.

Гиперпараметры для нашей модели с тремя сверточными потоками показаны в листинге 11.40, а также в нашем блокноте Jupyter *multi_convnet_sentiment_classifier.ipynb*.

---

[1] Эта модель conv-LSTM приблизилась к точности на проверочных данных и метрике ROC AUC нашей многослойной архитектуры Bi-LSTM, но каждой эпохе потребовалось на 82% меньше времени.

[2] К наиболее популярным аспектам непоследовательных моделей относятся: наличие нескольких входов или выходов (возможно, в разных слоях; например, модель может иметь дополнительный вход или выход в середине архитектуры), совместное использование активаций из одного слоя в множестве других слоев и поддержка ориентированных ациклических графов.

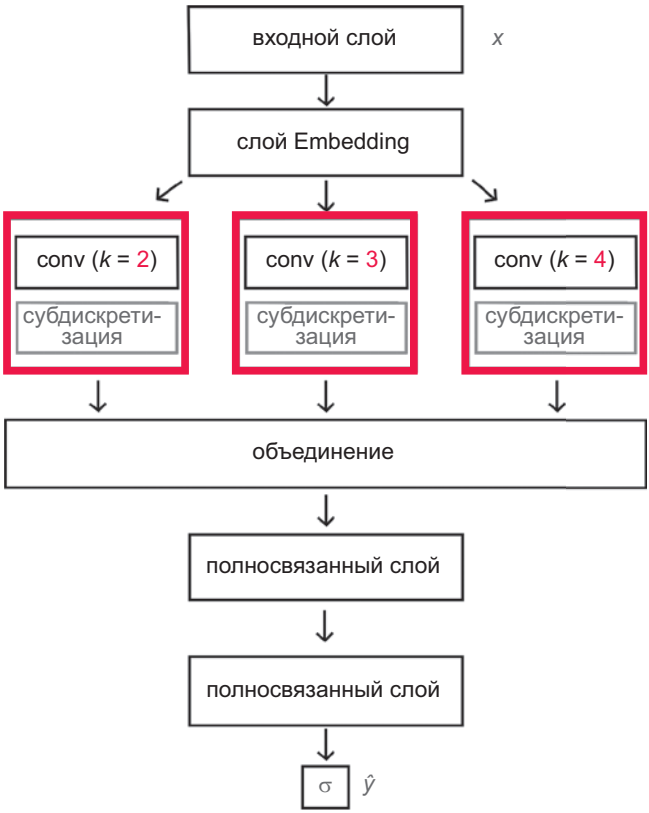


Рис. 11.27. Архитектура непоследовательной модели: три параллельных потока сверточных слоев — каждый с уникальной длиной фильтра (k = 2, k = 3 и k = 4) — получают входные данные от слоя, генерирующего векторные представления. Активации из всех трех потоков объединяются и передаются в пару полносвязанных слоев, следующих друг за другом на пути к сигмоидному выходному нейрону

Листинг 11.40. Гиперпараметры классификатора эмоциональной окраски с несколькими сверточными потоками

```
# имя каталога для сохранения результатов:
output_dir = 'model_output/multiconv'

# обучение:
epochs = 4
batch_size = 128

# параметры векторного пространства слов:
n_dim = 64
n_unique_words = 5000
max_review_length = 400
pad_type = trunc_type = 'pre'
drop_embed = 0.2
```

```
# архитектура сверточного слоя:
n_conv_1 = n_conv_2 = n_conv_3 = 256
k_conv_1 = 3
k_conv_2 = 2
k_conv_3 = 4

# архитектура полносвязанного слоя:
n_dense = 256
dropout = 0.2
```

Новые гиперпараметры относятся к трем сверточным слоям. Все три слоя имеют по 256 фильтров, но, как показано на рис. 11.27, они образуют параллельные потоки — каждый со своей длиной фильтра (k), от 2 до 4.

Архитектура модели с несколькими сверточными потоками показана в листинге 11.41.

Листинг 11.41. Архитектура классификатора эмоциональной окраски с несколькими сверточными потоками

```
from keras.models import Model
from keras.layers import Input, concatenate

# входной слой:
input_layer = Input(shape=(max_review_length,),
                    dtype='int16', name='input')

# слой векторного представления:
embedding_layer = Embedding(n_unique_words, n_dim,
                            name='embedding')(input_layer)
drop_embed_layer = SpatialDropout1D(drop_embed,
                                    name='drop_embed')(embedding_layer)

# три параллельных сверточных потока:
conv_1 = Conv1D(n_conv_1, k_conv_1,
                activation='relu', name='conv_1')(drop_embed_layer)
maxp_1 = GlobalMaxPooling1D(name='maxp_1')(conv_1)

conv_2 = Conv1D(n_conv_2, k_conv_2,
                activation='relu', name='conv_2')(drop_embed_layer)
maxp_2 = GlobalMaxPooling1D(name='maxp_2')(conv_2)

conv_3 = Conv1D(n_conv_3, k_conv_3,
                activation='relu', name='conv_3')(drop_embed_layer)
maxp_3 = GlobalMaxPooling1D(name='maxp_3')(conv_3)

# объединение активаций из трех потоков:
concat = concatenate([maxp_1, maxp_2, maxp_3])

# скрытые полносвязанные слои:
dense_layer = Dense(n_dense,
```

```
                    activation='relu', name='dense')(concat)
drop_dense_layer = Dropout(dropout, name='drop_dense')(dense_layer)
dense_2 = Dense(int(n_dense/4),
                activation='relu', name='dense_2')(drop_dense_layer)
dropout_2 = Dropout(dropout, name='drop_dense_2')(dense_2)

# сигмоидный выходной слой:
predictions = Dense(1, activation='sigmoid', name='output')(dropout_2)

# создание модели:
model = Model(input_layer, predictions)
```

Эта архитектура может показаться немного пугающей, если прежде вы не были знакомы с классом `Model` из библиотеки Keras, но следующее построчное описание поможет вам понять любые аспекты:

- Класс `Model` позволяет нам определить слой `Input()` отдельно, а не указывать его в виде аргумента `shape` первого скрытого слоя. Мы явно указали тип данных (`dtype`): 16-разрядные целые числа (`int16`), которые могут изменяться в диапазоне до 32 767, что соответствует максимальному индексу входных слов[1]. Как и для всех слоев в этой модели, определяем говорящее имя в аргументе `name`, чтобы при последующем выводе сводной информации о модели (вызовом `model.summary()`) мы смогли легко разобраться в ней.
- Каждый слой сохраняется в своей переменной, например `input_layer`, `embedding_layer` и `conv_2`. Мы будем использовать эти переменные при конструировании потоков данных в нашей модели.
- Наиболее примечательный аспект класса `Model`, хорошо знакомый разработчикам, которые используют функциональные языки программирования, — это имя переменной во втором наборе круглых скобок, следующих за каждым вызовом функции создания слоя. Оно определяет слой, выходные данные из которого будут поступать на вход данного слоя. Например, `(input_layer)` во втором наборе круглых скобок при создании `embedding_layer` указывает, что на вход слоя, создающего векторные представления слов, поступают выходные данные из входного уровня.
- Слои `Embedding()` и `SpatialDropout1D` принимают те же аргументы, что и выше в этой главе.
- Выходные данные из слоя `SpatialDropout1D` (переменная `drop_embed_layer`) передаются на вход трех отдельных параллельных сверточных слоев: `conv_1`, `conv_2` и `conv_3`.

[1] В данном случае максимальное значение индекса равно 5500, в соответствии с выбранными значениями гиперпараметров `n_unique_words` и `n_words_to_skip`.

- Как показано на рис. 11.27, каждый из трех сверточных потоков включает слой `Conv1D` (со своей длиной фильтра `k_conv`) и слой `GlobalMaxPooling1D`.
- Активации на выходе слоя `GlobalMaxPooling1D` в каждом из трех сверточных потоков объединяются в единый массив значений активации слоем `concatenate()`, который в качестве единственного аргумента принимает список входных данных (`[maxp_1, maxp_2, maxp_3]`).
- Объединенные активации сверточных потоков подаются на вход двух скрытых слоев `Dense()`, каждый из которых имеет соответствующий слой `Dropout()`. (Второй полносвязанный слой имеет в четыре раза меньше нейронов, чем первый, о чем свидетельствует аргумент `n_dense/4`.)
- Активации, выводимые нейроном сигмоидного слоя ($\hat{y}$), присваиваются переменной `predictions`.
- Наконец, класс `Model` связывает все слои вместе, принимая два аргумента: входной слой (то есть `input_layer`) и выходной (`predictions`).

В конечном итоге наша параллельная сетевая архитектура дала лишь небольшой прирост эффективности, но достаточный, чтобы вывести ее на первое место среди классификаторов эмоциональной окраски в этой главе (табл. 11.6). Как отмечено в нашем блокноте *multi_convnet_sentiment_classifier.ipynb*, наименьшие потери на проверочных данных были достигнуты во второй эпохе (0.262). Точность на проверочных данных в эту эпоху достигла 89.4%, а метрика ROC AUC — 96.2%, что на одну десятую процента лучше, чем в последовательной сверточной модели.

Таблица 11.6. Сравнение эффективности моделей классификаторов эмоциональной окраски

Модель	ROC AUC (%)
Полносвязанная	92.9
Сверточная	96.1
Простая рекуррентная	84.9
LSTM	92.8
Bi-LSTM	93.5
Многослойная Bi-LSTM	94.9
GRU	93.0
Conv-LSTM	94.5
Сверточная с несколькими потоками	96.2

ИТОГИ

В этой главе мы обсудили методы предварительной обработки данных на естественном языке, способы создания векторов слов из корпуса текстов и процедуру вычисления площади под кривой рабочей характеристики приемника. Во второй половине главы мы использовали эти знания для экспериментов с разнообразными моделями NLP глубокого обучения с целью классификации эмоциональной окраски отзывов к фильмам. Некоторые из этих моделей включали типы слоев, с которыми мы познакомились в предыдущих главах (полносвязанные и сверточные), но были среди них и модели, содержащие новые типы слоев из семейства RNN (LSTM и GRU), а также впервые в этой книге мы познакомились с моделями, имеющими непоследовательную архитектуру.

В табл. 11.6 представлена сводная информация с результатами наших экспериментов по классификации эмоциональной окраски. Мы думаем, что если бы наш набор данных на естественном языке был намного больше, архитектуры Bi-LSTM смогли бы превзойти сверточные сети.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым.

- параметры:
  - вес w;
  - смещение b;
- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
  - Linear;
- входной слой;
- скрытый слой;
- выходной слой;
- типы слоев:
  - плотный (полносвязанный);
  - softmax;
  - сверточный;
  - субдискретизации с объединением по максимальному значению;

  - преобразования в плоский массив;
  - создания векторных представлений;
  - рекуррентные;
  - (двунаправленные) LSTM;
  - объединения;
- функции стоимости (потерь):
  - квадратичная (среднеквадратичная ошибка);
  - перекрестная энтропия;
- прямое распространение;
- обратное распространение;
- нестабильность градиентов (в частности, затухание);
- метод Глоро инициализации весов;
- пакетная нормализация;
- прореживание;
- оптимизаторы:
  - стохастический градиентный спуск;
  - Adam;
- гиперпараметры оптимизаторов:
  - скорость обучения η;
  - размер пакета;
- word2vec

12

ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНЫЕ СЕТИ

Еще в главе 3 мы познакомились с моделями глубокого обучения, способными создавать новые уникальные визуальные образы — изображения, которые можно даже назвать искусством. В этой главе мы с помощью класса `Model` из библиотеки Keras объединим теорию из главы 3 со сверточными сетями из главы 10 и парой новых типов слоев, чтобы создать генеративно-состязательную сеть (Generative Adversarial Network, GAN), создающую изображения в стиле обычных эскизов, нарисованных человеком.

БАЗОВАЯ ТЕОРИЯ GAN

На верхнем уровне генеративно-состязательная сеть состоит из двух сетей глубокого обучения, состоящих в *состязательных* отношениях. Как показано на рис. 3.4 на примере трилобитов, одна сеть является *генератором* и производит поддельные изображения, а другая — *дискриминатором* и пытается отличить поддельные изображения от реальных. А теперь оставим трилобитов и рассмотрим схему, имеющую чуть более технический характер: генератору поручается на основе случайного входного шума создать ложное изображение, как показано слева на рис. 12.1. Дискриминатор — бинарный классификатор реальных и поддельных изображений — показан на рис. 12.1 справа. (Схема на этом рисунке очень упрощена и иллюстрирует лишь самые основы, но очень скоро мы с вами углубимся в технические детали.) В течение нескольких циклов обучения генератор совершенствуется в создании более убедительных подделок, а дискриминатор совершенствуется в способности отличать подделки. В ходе обучения обе модели состязаются, пытаясь превзойти друг друга, и при этом совершенствуют свои навыки. В конечном итоге обучение может завершиться тем, что генератор научится создавать подделки, убедительные не только для сети дискриминатора, но и для человека.

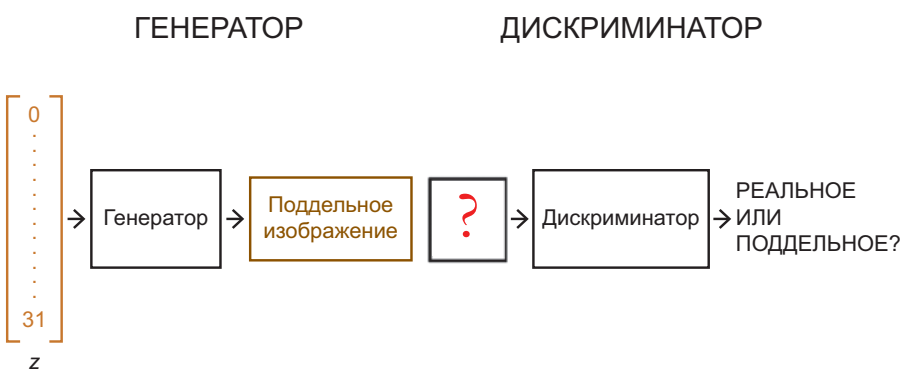


Рис. 12.1. Очень упрощенная схема двух моделей, составляющих типичную генеративно-состязательную сеть: генератора (слева) и дискриминатора (справа)

Обучение генеративно-состязательной сети заключается в выполнении двух противоположных (*состязательных!*) процессов:

1. *Обучение дискриминатора*: как показано на рис. 12.2, в этом процессе генератор создает поддельные изображения, то есть только выводит результат своей работы[1], а дискриминатор учится отличать поддельные изображения от реальных.
2. *Обучение генератора*: как показано на рис. 12.3, в этом процессе дискриминатор оценивает поддельные изображения, созданные генератором. Здесь *дискриминатор* только выводит свои результаты, а *генератор* использует их для *обучения* — в данном случае он учится обманывать дискриминатор, чтобы заставить его классифицировать поддельные изображения как реальные.

Таким образом, в каждом из этих процессов одна модель генерирует результат (поддельное изображение или класс, к которому относится это изображение), но не обучается, а другая использует этот результат, чтобы научиться еще лучше решать свою задачу.

В целом в ходе обучения генеративно-состязательной сети (GAN) происходит чередование обучения дискриминатора и генератора. Рассмотрим подробнее оба процесса и начнем с обучения дискриминатора (рис. 12.2):

- Генератор создает поддельные изображения (отмечены черным цветом), которые смешиваются с реальными и передаются в дискриминатор для обучения.
- Дискриминатор выводит прогноз ($\hat{y}$) — вероятность, что изображение является реальным.

[1] Вывод поддельных изображений — это только прямое распространение. Сюда не входит обучение модели (например, посредством обратного распространения).

ОБУЧЕНИЕ ДИСКРИМИНАТОРА

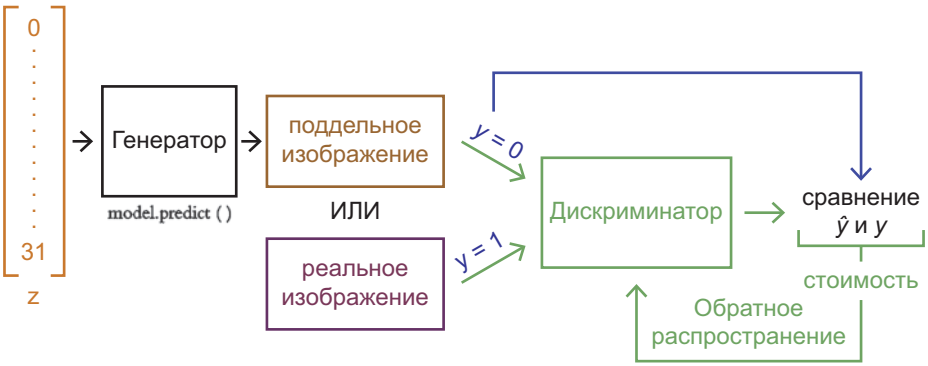


Рис. 12.2. Схема цикла обучения дискриминатора. Прямое распространение через генератор создает поддельные изображения. Они смешиваются с реальными изображениями в обучающий набор данных и используются для обучения дискриминатора. Пути обучения показаны зеленым цветом, а пути, когда дискриминатор не обучается, — черным. Синяя стрелка отмечает метки изображений, *y*

ОБУЧЕНИЕ ГЕНЕРАТОРА

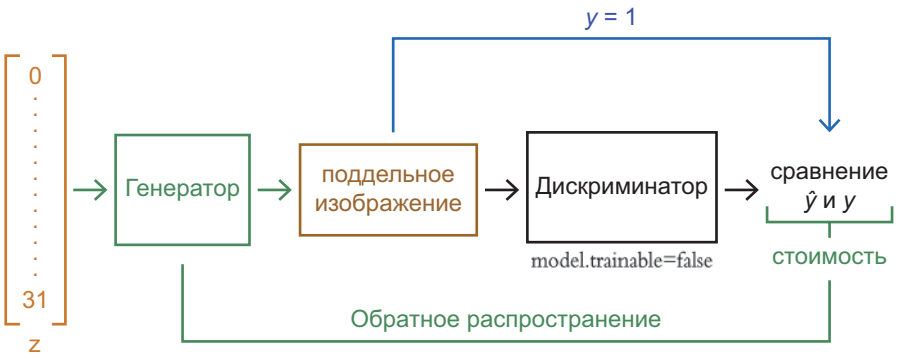


Рис. 12.3. Схема цикла обучения генератора. На этапе прямого распространения генератор создает поддельные изображения, а дискриминатор оценивает их. Генератор обучается на этапе обратного распространения. Как и на рис. 12.2, пути обучения показаны зеленым цветом, а пути, когда генератор не обучается, — черным. Синяя стрелка обозначает связь между изображением и его меткой *y*, которая в случае обучения генератора всегда равна 1

- Для прогнозов дискриминатора $\hat{y}$ вычисляется функция стоимости на основе перекрестной энтропии относительно истинных меток y.
- В ходе обратного распространения производится корректировка параметров дискриминатора (показано *зеленым* цветом) для минимизации стои-

мости, и модель учится еще лучше отличать реальные изображения от поддельных.

Обратите внимание, что во время обучения дискриминатора обучается только сеть дискриминатора; сеть генератора не участвует в обратном распространении и ничего не изучает.

Теперь рассмотрим обучение генератора, которое чередуется с процессом обучения дискриминатора (показан на рис. 12.3):

- Генератор получает вектор со случайным шумом z на входе[1] и создает поддельное изображение на выходе.
- Поддельные изображения, созданные генератором, передаются непосредственно в дискриминатор. Для этого процесса крайне важно попытаться обмануть дискриминатор, поэтому все поддельные изображения обозначаются как реальные ($y = 1$).
- Дискриминатор (на этапе вывода; показан черным цветом) выводит прогноз $\hat{y}$ — является ли данное изображение реальным или поддельным.
- Стоимость, вычисленная на основе перекрестной энтропии, используется для корректировки параметров сети генератора (показана *зеленым* цветом). В частности, генератор узнает, насколько убедительными являются его поддельные изображения для дискриминатора. Минимизируя эту стоимость, генератор учится создавать подделки, которые дискриминатор ошибочно распознает как реальные и которые могут показаться реальными даже человеку.

Таким образом, на этапе обучения генератора обучается *только* он сам. Далее в этой главе мы покажем, как зафиксировать параметры дискриминатора, чтобы на этапе обратного распространения корректировались лишь параметры генератора, а параметры дискриминатора оставались неизменными.

В начале обучения сети GAN генератор еще не знает, что он должен делать, поэтому, получая случайный шум на входе, генератор создает изображения случайного шума на выходе. Эти низкокачественные подделки резко отличаются от реальных изображений, содержащих комбинации признаков реальных изображений, и, как результат, дискриминатор с легкостью отличает реальные изображения от подделок. Однако постепенно генератор учится копировать некоторые структуры, характерные для реальных изображений. В конце концов он становится достаточно искусным, чтобы обмануть дискриминатор, но тот, в свою очередь, выявляет все более сложные и детализированные признаки реальных изображений, и перехитрить его становится все сложнее. В процессе

[1] Вектор z со случайным шумом соответствует вектору *скрытого пространства*, описанному в главе 3 (см. рис. 3.4), и он не связан с переменной z на рис. 6.8 с формулой $w \cdot x + b$. Мы познакомимся с ним поближе далее в этой главе.

такого поочередного обучения генератора и дискриминатора генератор учится создавать все более убедительные изображения. В какой-то момент две состязающиеся модели оказываются в тупике: они достигают пределов возможностей своих архитектур, и обучение обеих сторон останавливается[1].

По завершении обучения дискриминатор отбрасывается, и остается только генератор как конечный продукт. Обученному генератору можно передать любой случайный шум, и он будет выводить изображения, соответствующие стилю изображений, на которых обучалась состязательная сеть. В этом смысле генеративный потенциал сети GAN можно считать *творческим*. Если генеративно-состязательную сеть обучить на большом наборе фотографий лиц знаменитостей, она сможет создавать убедительные фотографии «звезд», которых никогда не существовало. Как показано на рис. 3.4, передавая конкретные значения z в такой генератор, можно задавать конкретные координаты в скрытом пространстве GAN и получать изображения лиц знаменитостей с любыми желаемыми атрибутами, такими как определенный возраст, пол или тип очков. В этой главе мы будем обучать сеть GAN на наборе данных, состоящем из набросков, нарисованных людьми, поэтому наша сеть научится создавать новые рисунки, которые прежде никогда не возникали в человеческом разуме. Не пропустите раздел, где подробно обсуждаются конкретные архитектуры генератора и дискриминатора. А сейчас мы расскажем вам, как загрузить и подготовить обучающие данные.

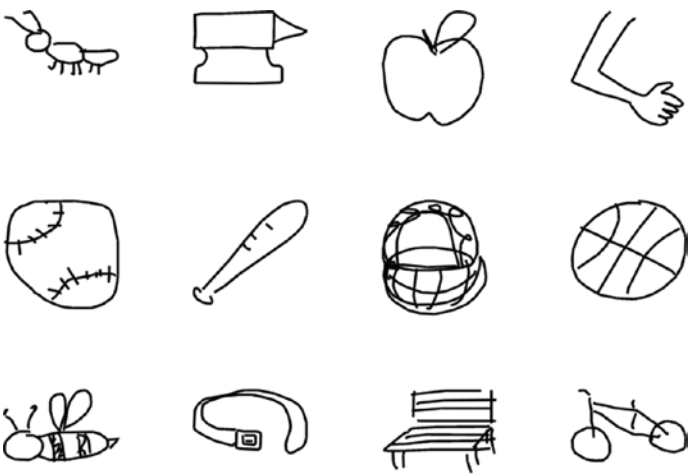


Рис. 12.4. Примеры набросков, нарисованных людьми, которые играли в онлайн-игру Quick, Draw! Бейсбольные биты, корзины и пчелы — ну и ну!

[1] Более сложные сети генераторов и дискриминаторов будут выявлять более сложные признаки и создавать более реалистичные изображения. Однако иногда такая сложность просто *не нужна*, и, конечно же, обучать такие модели будет сложнее.

НАБОР ДАННЫХ QUICK, DRAW!

В конце главы 1 мы пригласили вас сыграть в онлайн-игру Quick, Draw![1] Если вы последовали нашему приглашению, значит, вы тоже внесли свой вклад в самый большой в мире набор набросков. На момент написания этих строк набор данных Quick, Draw! включал 50 миллионов рисунков в 345 категориях. Примеры набросков из 12 таких категорий представлен на рис. 12.4, в том числе из категорий: *муравей*, *наковальня* и *яблоко*. Генеративно-состязательная сеть, которую мы создадим в этой главе, будет обучаться на изображениях яблок, но вы можете выбрать любую категорию, которая вам понравится. Можете даже попробовать обучить сеть сразу на нескольких категориях, если ищете приключений[2]!

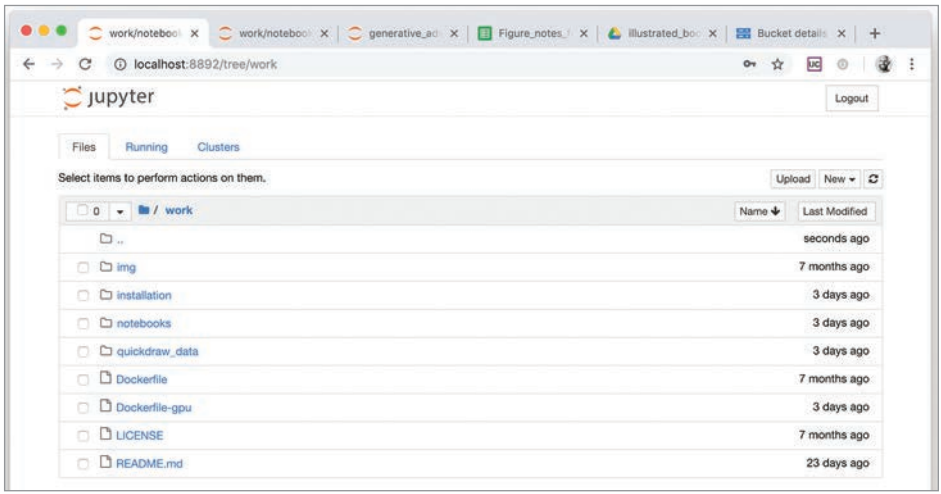


Рис. 12.5. Структура каталогов внутри контейнера Docker, где выполняется Jupyter. Мы поместили каталог quickdraw_data (с растровыми изображениями из набора Quick, Draw! в формате NumPy) на одном уровне с каталогом notebooks (содержащим все блокноты Jupyter для этой книги)

Набор данных Quick, Draw! в репозитории GitHub находится по адресу *bit.ly/QDrepository*. Они доступны в нескольких форматах, в том числе в виде исходных и необработанных изображений. Для получения относительно однородных данных мы советуем использовать предварительно обработанные, прошедшие

[1] quickdraw.withgoogle.com

[2] Если вы владеете мощной вычислительной техникой (как минимум с несколькими графическими процессорами), можете попробовать обучить сеть на данных сразу из всех 345 категорий набросков. Мы не пошли этим путем, поэтому результат действительно может оказаться весьма интересным.

процедуры центрирования и масштабирования. В частности, для простоты работы с данными в Python мы рекомендуем выбрать предварительно обработанные растровые изображения в формате NumPy[1].

Мы загрузили файл `apple.npy`, но вы можете выбрать любую другую категорию для обучения своей сети, какая вам понравится. На рис. 12.5 показано содержимое нашего рабочего каталога Jupyter с файлом данных из репозитория:

```
/deep-learning-illustrated/quickdraw_data/apples.npy
```

Вы можете сохранить данные в другом месте и изменить имя файла (например, если вы загрузили изображения из другой категории, отличной от категории *apples* — яблоки), но не забудьте в этом случае внести соответствующие изменения в код загрузки данных, приведенный в листинге 12.2.

Первый шаг, который должен показаться вам уже привычным, — это загрузка зависимостей. Зависимости для нашего блокнота *generative_adversarial_network.ipynb* представлены в листинге 12.1[2].

Листинг 12.1. Зависимости генеративно-состязательной сети

```
# для ввода и вывода данных:
import numpy as np
import os

# для глубокого обучения:
import keras
from keras.models import Model
from keras.layers import Input, Dense, Conv2D, Dropout
from keras.layers import BatchNormalization, Flatten
from keras.layers import Activation
from keras.layers import Reshape # новое!
from keras.layers import Conv2DTranspose, UpSampling2D # новое!
from keras.optimizers import RMSprop # новое!

# для создания графика:
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
```

Все эти зависимости уже рассматривались выше в этой книге, кроме трех новых типов слоев и оптимизатора RMSProp[3], которые мы рассмотрим, когда приступим к обсуждению архитектуры нашей модели.

---

[1] Эти конкретные данные доступны для загрузки по адресу bit.ly/QDdata.

[2] Архитектура нашей сети основана на архитектуре, предложенной Раулем Атензой (Rowel Atienza), которую вы можете найти в GitHub по адресу bit.ly/mnistGAN.

[3] Мы представили оптимизатор RMSProp в главе 9. Перейдите к разделу «Необычные оптимизаторы», чтобы освежить память.

А теперь вернемся к загрузке данных. Предположим, что вы настроили структуру каталогов, как было предложено выше, и загрузили файл `apple.npy`. Теперь эти данные можно загрузить, как показано в листинге 12.2.

Листинг 12.2. Загрузка данных Quick, Draw!

```
input_images = "../quickdraw_data/apple.npy"
data = np.load(input_images)
```

И снова, если вы создали другую структуру каталогов или выбрали другую категорию изображений из набора Quick, Draw!, измените соответственно путь в переменное `input_images`.

Обратившись к `data.shape`, можно узнать размеры двух измерений обучающих данных. Первое измерение — количество изображений. На момент написания этих строк в категории яблок имелось 145 000 изображений, но когда вы будете читать их, скорее всего, будет уже больше. Второе измерение — количество пикселов в каждом изображении. Это значение равно `784` и должно показаться вам знакомым, потому что, как и цифры в наборе MNIST, эти изображения имеют форму 28 × 28 пикселов.

Изображения из набора Quick, Draw! не только имеют те же размеры, что и цифры MNIST, но их пикселы точно так же представлены 8-разрядными целыми числами, то есть числами в диапазоне от 0 до 255. Убедиться в этом можно, проверив содержимое одного, например, 4243-го изображения, выполнив инструкцию `data[4242]`. Поскольку данные находятся в одномерном массиве, вы увидите только последовательность чисел. Данные следует переформатировать:

```
data = data/255
data = np.reshape(data,(data.shape[0],28,28,1))
img_w,img_h = data.shape[1:3]
```

Вот что делает каждая строка в этом фрагменте кода:

- Значения пикселов делятся на `255`, чтобы привести их в диапазон от 0 до 1, как мы делали это при работе с цифрами MNIST[1].
- Первый скрытый слой сети дискриминатора будет состоять из двумерных сверточных фильтров, поэтому мы преобразуем изображения из одномерных массивов с 784 пикселами в двумерные матрицы 28 × 28 пикселов. Для этого вызывается метод `reshape()` из пакета NumPy. Обратите внимание, что четвертое измерение равно `1`, потому что изображения являются монохромными; если бы изображения были полноцветными, оно имело бы размер `3`.
- Ширина изображения (`img_w`) и его высота (`img_h`) сохраняются для использования в будущем.

[1] См. сноску перед листингом 5.4, где объясняется, зачем необходимо масштабирование.

На рис. 12.6 показан пример того, как выглядят переформатированные данные. Мы получили это изображение 4243-го наброска из категории «яблоки», выполнив следующий код:

```
plt.imshow(data[4242,:,:,0], cmap='Greys')
```

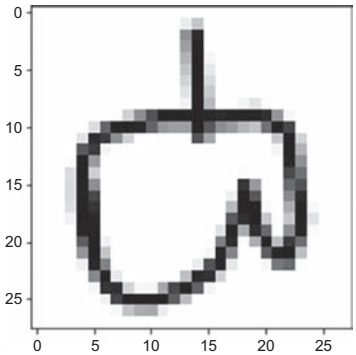


Рис. 12.6. Пример 4243-го наброска из категории «яблоки» в наборе Quick, Draw!

СЕТЬ ДИСКРИМИНАТОРА

Наш дискриминатор — это довольно простая сверточная сеть, включающая слои `Conv2D`, подробно описанные в главе 10, и класс `Model`, представленный в конце главы 11. Взгляните на код в листинге 12.3.

Листинг 12.3. Архитектура модели дискриминатора

```
def build_discriminator(depth=64, p=0.4):

    # Определение входов
    image = Input((img_w,img_h,1))

    # Сверточные слои
    conv1 = Conv2D(depth*1, 5, strides=2,
                   padding='same', activation='relu')(image)
    conv1 = Dropout(p)(conv1)

    conv2 = Conv2D(depth*2, 5, strides=2,
                   padding='same', activation='relu')(conv1)
    conv2 = Dropout(p)(conv2)

    conv3 = Conv2D(depth*4, 5, strides=2,
                   padding='same', activation='relu')(conv2)
    conv3 = Dropout(p)(conv3)

    conv4 = Conv2D(depth*8, 5, strides=1,
                   padding='same', activation='relu')(conv3)
    conv4 = Flatten()(Dropout(p)(conv4))
```

```
    # выходной слой
    prediction = Dense(1, activation='sigmoid')(conv4)

    # Определение модели
    model = Model(inputs=image, outputs=prediction)

    return model
```

Впервые в этой книге вместо непосредственного создания архитектуры модели мы определили функцию (`build_discriminator`), возвращающую сконструированный объект модели. Давайте подробно рассмотрим устройство этой модели, которая изображена на рис. 12.7 и представлена в листинге 12.3:

- Входные изображения имеют размер 28 × 28 пикселов. Эта информация передается во входной слой через переменные `img_w` и `img_h`.
- В модели имеется четыре скрытых слоя, и все они сверточные.
- Количество сверточных фильтров удваивается в каждом следующем слое, то есть если первый скрытый слой имеет 64 сверточных фильтра (и выводит карту активации с глубиной 64), то четвертый имеет уже 512 сверточных фильтров (и карту активации с глубиной 512)[1].
- Размер фильтра 5 × 5 определяется как константа[2].
- Длина шага для первых трех сверточных слоев составляет 2 × 2, то есть высота и ширина карты активаций уменьшаются примерно вдвое в каждом следующем слое (вспомните уравнение 10.3). Длина шага для последнего сверточного слоя равна 1 × 1, поэтому карта активаций, которую он выводит, имеет ту же высоту и ширину, что и на входе (4 × 4).
- К каждому сверточному слою применяется прореживание на уровне 40% (`p=0.4`).
- Мы преобразуем трехмерную карту активации, полученную на выходе конечного сверточного слоя, в плоский массив, чтобы передать ее в полносвязанный слой.
- Так же, как модели оценки эмоциональной окраски отзывов к фильмам в главе 11, определение реальных и поддельных изображений является задачей бинарной классификации, поэтому наш (полносвязанный) выходной слой состоит из единственного сигмоидного нейрона.

---

[1] Увеличение числа фильтров влечет увеличение параметров и усложнение модели, а также способствует повышению четкости изображений, создаваемых сетью. Эти значения дают достаточно хорошие результаты в этом примере.

[2] До сих пор мы обычно использовали фильтры с размерами 3 × 3, однако генеративно-состязательные сети могут выиграть от использования фильтров немного большего размера, особенно в первых слоях.

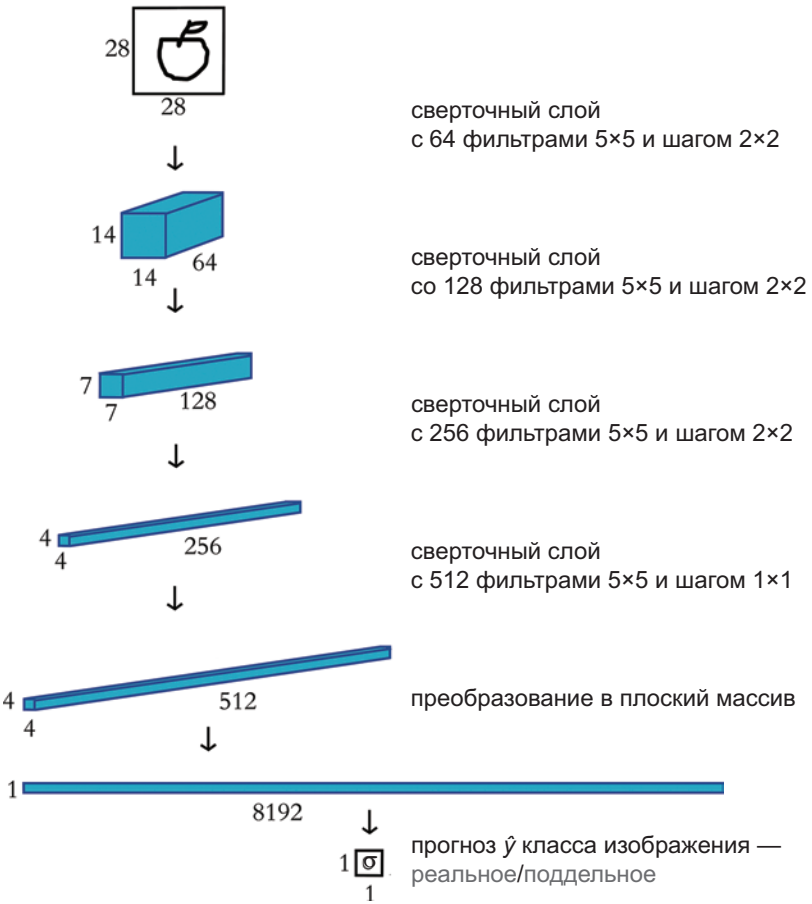


Рис. 12.7. Схематическое представление сети дискриминатора, определяющей, является ли входное изображение реальным (в данном случае изображением яблока, нарисованным от руки, из набора данных Quick, Draw!) или поддельным (созданным генератором изображений)

Чтобы сконструировать дискриминатор, вызываем функцию `build_discriminator` без аргументов:

```
discriminator = build_discriminator()
```

Сводную информацию об архитектуре модели можно вывести с помощью метода `summary` модели, в которой можно увидеть, что модель имеет в общей сложности 4.3 миллиона параметров, большинство из которых (76%) находятся в конечном сверточном слое.

В листинге 12.4 приводится код, осуществляющий компиляцию дискриминатора.

Листинг 12.4. Компиляция сети дискриминатора

```
discriminator.compile(loss='binary_crossentropy',
                      optimizer=RMSprop(lr=0.0008,
                                        decay=6e-8,
                                        clipvalue=1.0),
                      metrics=['accuracy'])
```

Рассмотрим листинг 12.4 подробнее.

- Так же, как в главе 11, мы используем бинарную функцию стоимости на основе перекрестной энтропии, потому что дискриминатор является моделью бинарной классификации.
- RMSprop, представленный в главе 9, — это один из «необычных оптимизаторов», служащий альтернативой для Adam[1].
- Скорость затухания (`decay`, ρ) в оптимизаторе RMSprop является гиперпараметром, как описывается в главе 9.
- Наконец, `clipvalue` — это гиперпараметр, не позволяющий градиенту обучения (отношению частной производной между стоимостью и значениями параметров во время стохастического градиентного спуска) превысить это значение; таким образом, `clipvalue` явно ограничивает взрыв градиента (см. главу 9). Данное конкретное значение `1.0` является типичным.

СЕТЬ ГЕНЕРАТОРА

В отличие от сети дискриминатора, имеющей довольно знакомую нам архитектуру, сеть генератора имеет ряд черт, которые еще не были описаны в этой книге. Схема модели генератора показана на рис. 12.8.

Мы называем генератор сетью *de*CNN («развертывающей» нейронной сетью), потому что он включает *«развертывающие» слои* (также известные как слои *convTranspose*), которые выполняют функцию, противоположную типичным сверточным слоям. Вместо выявления признаков и вывода карты активаций, отражающей, где эти признаки встречаются в изображении, развертывающие слои принимают карту активаций и размещают признаки в пространстве выходных данных. Первым делом генеративная сеть преобразует входной шум (одномерный вектор) в двумерный массив, который может использоваться развертывающими слоями. Пропуская шум через несколько развертывающих слоев, генератор преобразует его в поддельные изображения.

---

[1] Ян Гудфеллоу (Ian Goodfellow) и его коллеги опубликовали первую статью с описанием GAN в 2014 году. В то время оптимизатор RMSProp уже вошел в моду (исследователи Кингма и Ба опубликовали статью с описанием Adam в 2014 году, и с того момента его популярность возросла). Вы можете попробовать немного настроить гиперпараметры или заменить RMSProp на Adam и получить тот же эффект.

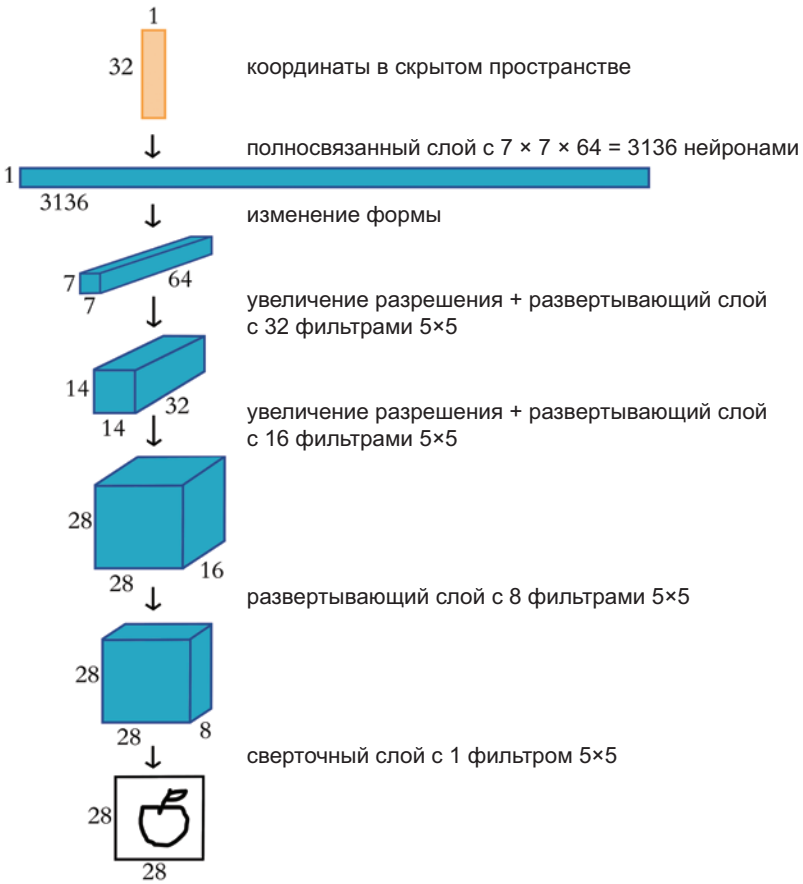


Рис. 12.8. Схематическое представление сети генератора, которая принимает шум (в данном случае представляющий 32 измерения скрытого пространства) и выводит изображение размером 28 × 28 пикселов. После обучения в составе состязательной сети они должны напоминать изображения из обучающего набора данных (в данном случае яблоки, нарисованные от руки)

В листинге 12.5 показан код, конструирующий модель генератора.

Листинг 12.5. Архитектура модели генератора

```
z_dimensions = 32
def build_generator(latent_dim=z_dimensions,
                    depth=64, p=0.4):
    # Определение входов
    noise = Input((latent_dim,))

    # Первый полносвязанный слой
    dense1 = Dense(7*7*depth)(noise)
    dense1 = BatchNormalization(momentum=0.9)(dense1)
```

```
dense1 = Activation(activation='relu')(dense1)
dense1 = Reshape((7,7,depth))(dense1)
dense1 = Dropout(p)(dense1)

# Развертывающие слои
conv1 = UpSampling2D()(dense1)
conv1 = Conv2DTranspose(int(depth/2),
                        kernel_size=5, padding='same',
                        activation=None,)(conv1)
conv1 = BatchNormalization(momentum=0.9)(conv1)
conv1 = Activation(activation='relu')(conv1)

conv2 = UpSampling2D()(conv1)
conv2 = Conv2DTranspose(int(depth/4),
                        kernel_size=5, padding='same',
                        activation=None,)(conv2)
conv2 = BatchNormalization(momentum=0.9)(conv2)
conv2 = Activation(activation='relu')(conv2)

conv3 = Conv2DTranspose(int(depth/8),
                        kernel_size=5, padding='same',
                        activation=None,)(conv2)
conv3 = BatchNormalization(momentum=0.9)(conv3)
conv3 = Activation(activation='relu')(conv3)

# Выходной слой
image = Conv2D(1, kernel_size=5, padding='same',
               activation='sigmoid')(conv3)

# Определение модели
model = Model(inputs=noise, outputs=image)

return model
```

Рассмотрим эту архитектуру поближе:

- Задаем число измерений во входном векторе шума (`z_dimensions`) равным 32. При настройке этого гиперпараметра мы следовали тому же совету, который давали для выбора количества измерений в векторном пространстве слов в главе 11: большее число измерений в векторе шума позволит сохранять больше информации и, следовательно, может улучшить качество поддельных изображений; однако при этом увеличивается вычислительная сложность. Мы обычно рекомендуем экспериментировать с этим гиперпараметром, выбирая для него значения, кратные 2.
- По аналогии с моделью дискриминатора (листинг 12.3) мы решили завернуть архитектуру генератора в функцию.
- Входными данными является массив случайных шумов с длиной `latent_dim`, которая в этом случае равна 32.
- Первый скрытый слой — это полносвязанный слой, позволяющий гибко отображать входное скрытое пространство в последующие пространствен-

ные (развертывающие) скрытые слои. 32 входных измерения отображаются в 3136 нейронов в полносвязанном слое, который выводит одномерный массив активаций. Далее эти активации преобразуются в карту активаций $7 \times 7 \times 64$. Этот полносвязанный слой — единственный в генераторе, к которому применяется прореживание.

- Сеть содержит три развертывающих слоя (`Conv2DTranspose`). Первый имеет 32 фильтра, и это число уменьшается вдвое в последующих двух слоях[1]. Несмотря на уменьшение *количества* фильтров, их *размер* увеличивается благодаря слоям повышения дискретизации (`UpSampling2D`). Каждый раз, когда применяется повышение дискретизации (с параметрами по умолчанию, как здесь), высота и ширина карты активаций удваиваются[2]. Во всех трех развертывающих слоях выбраны:
  - размер фильтра 5×5;
  - размер шага 1×1 (по умолчанию);
  - режим дополнения `same`, чтобы сохранить размеры карт активаций после развертывания;
  - функция активации ReLU;
  - пакетная нормализация (для регуляризации).
- Роль выходного слоя играет сверточный слой, который свертывает карты активаций $28 \times 28 \times 8$ в единственное *изображение* $28 \times 28 \times 1$. Сигмоидная функция активации на этом последнем шаге гарантирует изменение значений пикселов в диапазоне от `0` до `1`, точно так же, как данные из реальных изображений, которые вводятся в дискриминатор отдельно.

Так же, как мы это сделали с сетью дискриминатора, для создания генератора вызываем функцию `build_generator` без аргументов:

```
generator = build_generator()
```

Вызов метода `summary` модели показывает, что генератор имеет всего 177 000 обучаемых параметров, что составляет всего 4% от числа параметров в дискриминаторе.

СОСТЯЗАТЕЛЬНАЯ СЕТЬ

Объединив обучающие процессы на рис. 12.2 и 12.3, мы приходим к схеме на рис. 12.9. Выполняя примеры кода в этой главе, мы достигли следующего:

---

[1] По аналогии со сверточными слоями количество фильтров в слое соответствует количеству срезов (глубине) карты активаций, получающейся на выходе слоя.

[2] Грубо говоря, это делает повышение дискретизации обратным процессу субдискретизации.

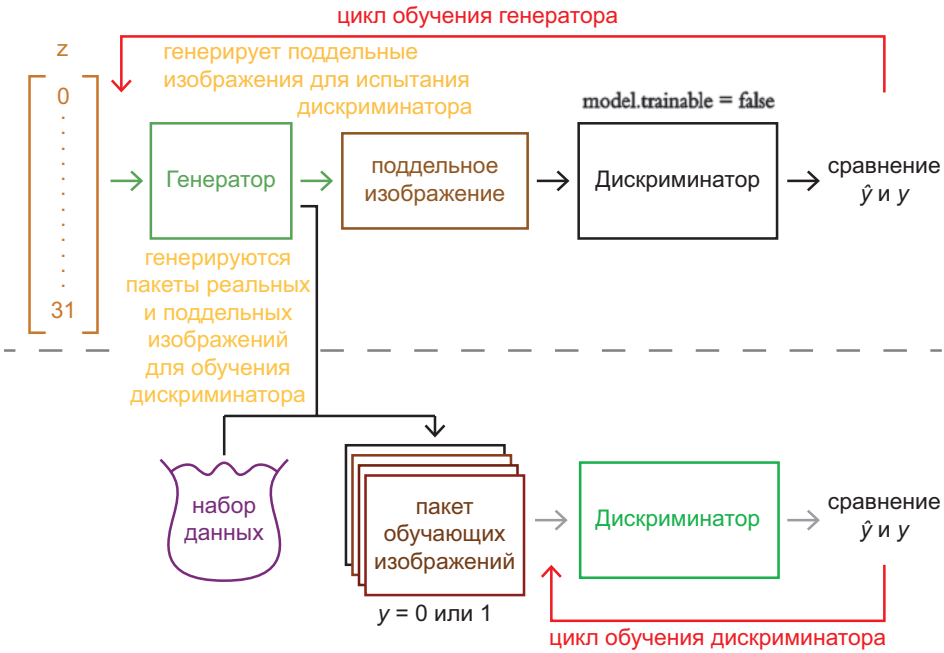


Рис. 12.9. Здесь изображена обобщенная схема всей состязательной сети. Горизонтальная пунктирная линия отделяет процессы обучения генератора и дискриминатора. Зеленые линии обозначают пути обучения, а черные — пути вывода результатов. Красные стрелки вверху и внизу обозначают шаг обратного распространения в каждом соответствующем обучающем процессе

- Для *обучения дискриминатора* (рис. 12.2) мы создали модель дискриминатора и скомпилировали ее: она готова учиться на реальных и поддельных изображениях различать эти два класса.
- Для *обучения генератора* (рис. 12.3) мы создали модель генератора, но чтобы подготовить ее к обучению, ее нужно скомпилировать как часть более крупной состязательной сети.

Объединим сети генератора и дискриминатора, чтобы получить состязательную сеть, как показано в листинге 12.6.

Листинг 12.6. Архитектура состязательной модели

```
z = Input(shape=(z_dimensions,))
img = generator(z)
discriminator.trainable = False
pred = discriminator(img)
adversarial_model = Model(z, pred)
```

Рассмотрим этот код поближе:

- Вызовом `Input()` мы определяем входной слой модели `z` как массив случайного шума с длиной 32.
- Передаем `z` в генератор `generator` и получаем изображение 28 × 28, которое сохраняем в переменной `img`.
- Перед началом цикла обучения генератора необходимо зафиксировать параметры сети дискриминатора (см. рис. 12.3), поэтому мы присваиваем атрибуту `trainable` дискриминатора значение `False`.
- Передаем поддельное изображение `img` в сеть дискриминатора с зафиксированными параметрами и получаем результат (`pred`) его классификации.
- Наконец, используя класс `Model` из библиотеки Keras, создаем состязательную модель. Указываем, что входом этой модели является слой `z`, а выходным значением — `pred`, функциональный API автоматически определяет, что состязательная сеть состоит из генератора, передающего `img` в зафиксированный дискриминатор.

В листинге 12.7 показано, как скомпилировать эту модель.

Листинг 12.7. Компиляция состязательной сети

```
adversarial_model.compile(loss='binary_crossentropy',
                          optimizer=RMSprop(lr=0.0004,
                                            decay=3e-8,
                                            clipvalue=1.0),
                          metrics=['accuracy'])
```

Аргументы метода `compile()` совпадают с теми, которые использовались для дискриминатора (см. листинг 12.4), с той лишь разницей, что скорость обучения и затухание градиента в оптимизаторе уменьшены вдвое. Важно соблюдать довольно тонкий баланс между скоростью обучения дискриминатора и генератора, чтобы генеративно-состязательная сеть могла создавать правдоподобные поддельные изображения. Настраивая гиперпараметры оптимизатора модели дискриминатора при ее компиляции, можно обнаружить, что также необходимо настроить гиперпараметры состязательной модели, чтобы получить более или менее удовлетворительные выходные изображения.

Важно отметить один хитрый аспект процесса обучения генеративно-состязательной сети: *одни и те же* параметры (веса) дискриминатора используются во время обучения самого дискриминатора и всей состязательной сети. Параметры дискриминатора фиксируются, только когда он находится в составе состязательной модели. Поэтому на этапе обучения дискриминатора его веса корректируются в ходе обратного распространения, и модель учится различать реальные и поддельные изображения. Состязательная модель, напротив, была скомпилирована с зафиксированным дискриминатором. Этот дискриминатор является точно такой же моделью с теми же весами, но когда обучающаяся модель обнаруживает, что *не должна* обновлять веса дискриминатора, она обновляет только веса генератора.

ОБУЧЕНИЕ ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНОЙ СЕТИ

Чтобы обучить нашу сеть GAN, вызовем функцию, метко названную `train`, которая представлена в листинге 12.8.

Листинг 12.8. Обучение сети GAN

```
def train(epochs=2000, batch=128, z_dim=z_dimensions):

    d_metrics = []
    a_metrics = []

    running_d_loss = 0
    running_d_acc = 0
    running_a_loss = 0
    running_a_acc = 0

    for i in range(epochs):

        # образцы реальных изображений:
        real_imgs = np.reshape(
            data[np.random.choice(data.shape[0],
                                  batch,
                                  replace=False)],
            (batch,28,28,1))

        # сгенерировать поддельные изображения:
        fake_imgs = generator.predict(
            np.random.uniform(-1.0, 1.0,
                              size=[batch, z_dim]))

        # объединить изображения в единый массив для передачи
        # дискриминатору:
        x = np.concatenate((real_imgs,fake_imgs))

        # присвоить метки y для дискриминатора:
        y = np.ones([2*batch,1])
        y[batch:,:] = 0

        # обучить дискриминатор:
        d_metrics.append(
            discriminator.train_on_batch(x,y)
        )
        running_d_loss += d_metrics[-1][0]
        running_d_acc += d_metrics[-1][1]

        # входной шум для передачи в состязательную сеть и метки y
        # "реальное":
        noise = np.random.uniform(-1.0, 1.0,
                                  size=[batch, z_dim])
        y = np.ones([batch,1])

        # обучение состязательной сети:
        a_metrics.append(
            adversarial_model.train_on_batch(noise,y)
        )
```

```
        running_a_loss += a_metrics[-1][0]
        running_a_acc += a_metrics[-1][1]

        # периодически выводить информацию о том, как движется обучение,
        # и получающиеся поддельные изображения:
        if (i+1)%100 == 0:

            print('Epoch #{}'.format(i))
            log_mesg = "%d: [D loss: %f, acc: %f]" % \
            (i, running_d_loss/i, running_d_acc/i)
            log_mesg = "%s [A loss: %f, acc: %f]" % \
            (log_mesg, running_a_loss/i, running_a_acc/i)
            print(log_mesg)

            noise = np.random.uniform(-1.0, 1.0,
                                      size=[16, z_dim])
            gen_imgs = generator.predict(noise)

            plt.figure(figsize=(5,5))

            for k in range(gen_imgs.shape[0]):
                plt.subplot(4, 4, k+1)
                plt.imshow(gen_imgs[k, :, :, 0],
                           cmap='gray')
                plt.axis('off')

            plt.tight_layout()
            plt.show()

    return a_metrics, d_metrics

# Обучить сеть GAN:
a_metrics_complete, d_metrics_complete = train()
```

Это был самый большой фрагмент кода в книге, поэтому разберем его подробнее:

- Два пустых списка (например, `d_metrics`) и четыре переменных (например, `running_d_loss`), которым присваивается значение `0`, предназначены для хранения оценок потерь и точности дискриминатора (`d`) и состязательной сети (`a`) во время их обучения.
- Для обучения в течение заданного числа эпох `epochs` используется цикл `for`. Обратите внимание: для обозначения цикла обучения разработчики GAN часто используют термин *эпоха*, хотя правильнее было бы называть его *пакетом*, потому что во время каждой итерации цикла `for` мы будем отбирать только `128` рисунков яблок из набора, включающего сотни тысяч.
- В каждой эпохе поочередно обучаются дискриминатор и генератор.
- Для обучения дискриминатора (как показано на рис. 12.2) мы:
  - Выбираем 128 реальных изображений.
  - Генерируем 128 поддельных изображений, создав векторы с шумом (`z`, заполненные значениями, которые равномерно выбираются из диапазо-

на [–1.0, 1.0]) и передав их в метод `predict` модели генератора. Обратите внимание, что при использовании метода `predict` генератор *только генерирует* изображения, не корректируя своих параметров.

- Объединяем реальные и фиктивные изображения в единый пакет в переменной `x`, который затем будет передан на вход дискриминатора.
- Создаем массив `y` с метками $y = 1$ для реальных изображений и $y = 0$ — для поддельных, который будет использоваться для обучения дискриминатора.
- Для обучения дискриминатора входные данные `x` и метки `y` передаются в вызов метода `train_on_batch` модели `discriminator`.
- После каждого цикла обучения метрики потерь и точности на обучающих данных добавляются в конец списка `d_metrics`.

❍ Для обучения генератора (как показано на рис. 12.3) мы:

- Передаем векторы со случайным шумом (хранящиеся в переменной `noise`) и массив (`y`), содержащий только метки, утверждающие реальность изображений (то есть $y = 1$), в метод `train_on_batch` состязательной модели.
- Генератор, компонент состязательной модели, преобразует входной шум в поддельные изображения, которые автоматически передаются на вход компонента-дискриминатора.
- Поскольку параметры дискриминатора зафиксированы во время обучения состязательной модели, дискриминатор просто сообщает, что он думает о реальности входящих изображений. Несмотря на то что генератор выводит поддельные изображения, они маркируются как реальные ($y = 1$), а стоимость на основе перекрестной энтропии используется во время обратного распространения для обновления весов модели генератора. Минимизируя эту стоимость, генератор учится создавать поддельные изображения, которые дискриминатор будет классифицировать как реальные.
- После каждого цикла обучения состязательной модели ее метрики потерь и точности добавляются в конец списка `a_metrics`.

❍ После каждых 100 эпох мы:

- Выводим порядковый номер эпохи.
- Выводим сообщение, включающее метрики потерь и точности дискриминатора и состязательной модели.
- Генерируем 16 случайных векторов с шумом и используем метод `predict` генератора, чтобы создать поддельные изображения и сохранить их в `gen_imgs`.
- Выводим 16 поддельных изображений в виде сетки 4×4, чтобы получить возможность визуально контролировать изменение качества изображений, создаваемых генератором в ходе обучения.

- По завершении функция `train` возвращает списки с метриками состязательной модели и модели дискриминатора (`a_metrics` и `d_metrics` соответственно).

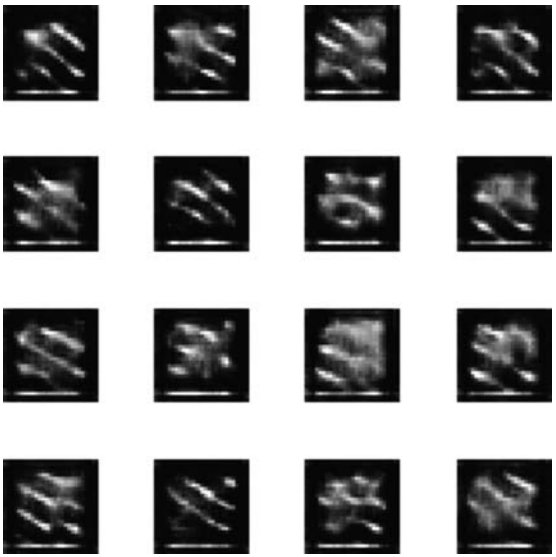


Рис. 12.10. Поддельные рисунки яблок, сгенерированные сетью GAN после 100 эпох обучения

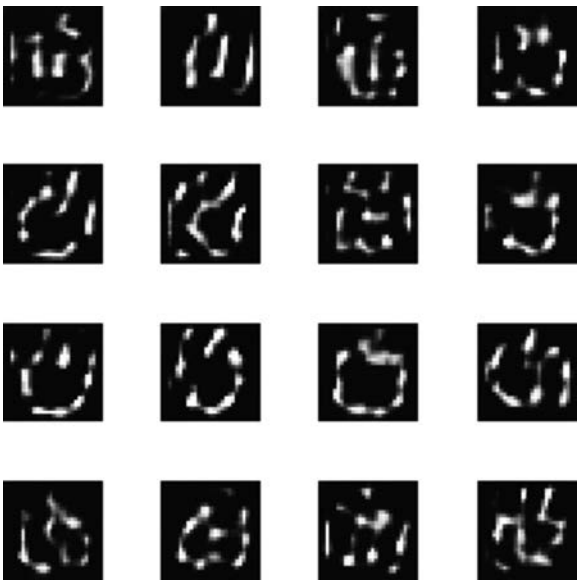


Рис. 12.11. Поддельные рисунки яблок после 200 эпох обучения

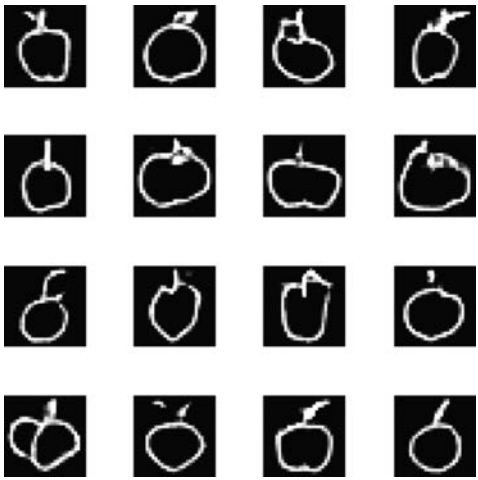


Рис. 12.12. Поддельные рисунки яблок после 1000 эпох обучения

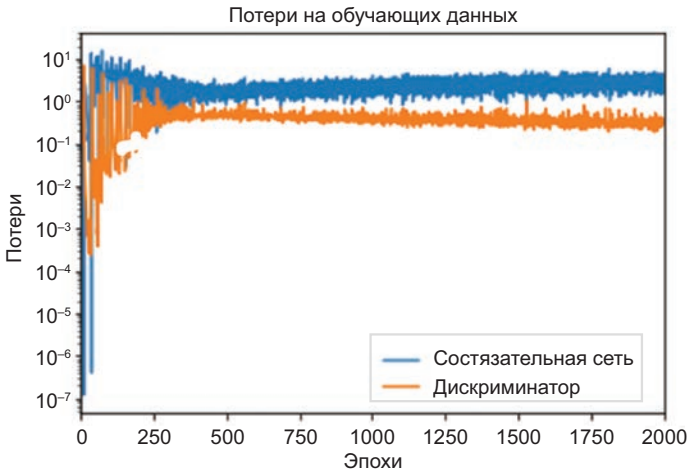


Рис. 12.13. Изменение потерь сети GAN на обучающих данных

- Наконец, мы вызываем функцию `train` и сохраняем метрики в переменных `a_metrics_complete` и `d_metrics_complete`.

После 100 циклов (эпох) обучения (рис. 12.10) наша сеть GAN начинает генерировать изображения, имеющие некоторые зачатки структуры, характерной для набросков, но яблоки в них пока не различаются. Однако после 200 циклов (рис. 12.11) в изображениях начинают появляться некоторые *черты яблок*. Спустя еще несколько сотен циклов обучения сеть начинает генерировать вполне убедительные подделки набросков яблок (рис. 12.12). А после 2000 раундов

она выдает изображения, которые являются практически «произведениями машинного искусства» и были представлены еще в конце главы 3 (см. рис. 3.9).

В конец нашего блокнота *generative_adversarial_network.ipynb* мы добавили код, который показан в листингах 12.9 и 12.10; он создает графики потерь (рис. 12.13) и точности (рис. 12.14) сети GAN на обучающих данных. На них ясно видно, что потери состязательной модели снижаются по мере улучшения качества поддельных рисунков яблок. Это вполне ожидаемо, потому что потери модели GAN связаны с ошибочной классификацией сетью дискриминатора поддельных изображений как реальных, и, как можно видеть на рис. 12.10, 12.11 и 12.12, чем дольше сеть обучается, тем реальнее становятся подделки. Когда генератор как компонент состязательной модели начал производить подделки более высокого качества, задача дискриминатора по распознаванию реальных и поддельных набросков усложнилась, в результате чего его потери росли в течение первых 300 эпох. Начиная с ~300-й эпохи дискриминатор постепенно совершенствовал свои навыки бинарной классификации, что соответствует плавному уменьшению потерь и увеличению точности на обучающих данных.

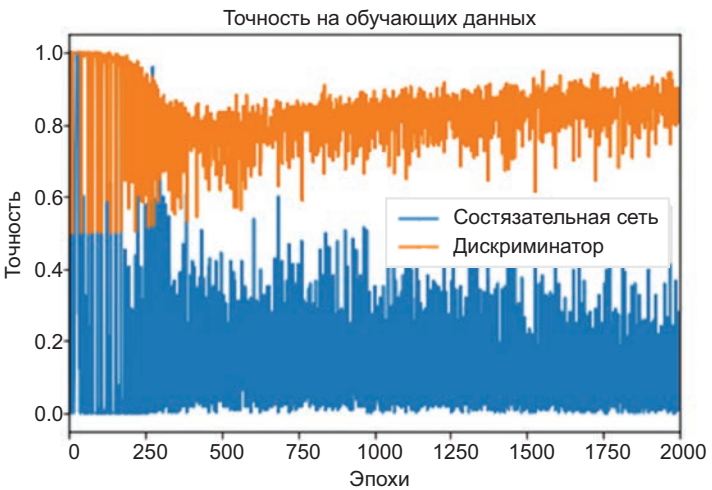


Рис. 12.14. Изменение точности сети GAN на обучающих данных

Листинг 12.9. Создание графика потерь GAN на обучающих данных

```
ax = pd.DataFrame(
    {
        'Adversarial': [metric[0] for metric in a_metrics_complete],
        'Discriminator': [metric[0] for metric in d_metrics_complete],
    }
).plot(title='Training Loss', logy=True)

ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
```

Листинг 12.10. Создание графика точности GAN на обучающих данных

```
ax = pd.DataFrame(
    {
        'Adversarial': [metric[1] for metric in a_metrics_complete],
        'Discriminator': [metric[1] for metric in d_metrics_complete],
    }
).plot(title='Training Accuracy')

ax.set_xlabel("Epochs")
ax.set_ylabel("Accuracy")
```

ИТОГИ

В этой главе мы рассмотрели теоретические основы генеративно-состязательных сетей и познакомились с новыми типами слоев (развертывающих и повышающих дискретизацию), построили сети дискриминатора и генератора, а затем объединили их, сформировав состязательную сеть. В результате попеременного обучения дискриминатора и генератора как компонента состязательной модели сеть GAN научилась создавать новые «рисунки» яблок.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым.

- параметры:
  - вес *w;*
  - смещение *b;*
- активация *a;*
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
  - Linear;
- входной слой;
- скрытый слой;
- выходной слой;
- типы слоев:
  - плотный (полносвязанный);
  - softmax;
  - сверточный;

  - развертывающий;
  - субдискретизации с объединением по максимальному значению;
  - повышения дискретизации;
  - преобразования в плоский массив;
  - создания векторных представлений;
  - рекуррентные;
  - (двунаправленные) LSTM;
  - объединения;
- функции стоимости (потерь):
  - квадратичная (среднеквадратичная ошибка);
  - перекрестная энтропия;
- прямое распространение;
- обратное распространение;
- нестабильность градиентов (в частности, затухание);
- метод Глоро инициализации весов;
- пакетная нормализация;
- прореживание;
- оптимизаторы:
  - стохастический градиентный спуск;
  - Adam;
- гиперпараметры оптимизаторов:
  - скорость обучения η;
  - размер пакета;
- word2vec;
- компоненты GAN:
  - сеть дискриминатора;
  - сеть генератора;
  - состязательная сеть.

13

ГЛУБОКОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ

В главе 4 мы познакомились с парадигмой обучения с подкреплением (дополняющей обучение с учителем и без учителя), в котором агент (например, алгоритм) выполняет последовательность действий в окружении. Окружения — имитации или реальный мир — могут быть чрезвычайно сложными и быстро изменяющимися, требующими, чтобы агенты могли соответствующим образом адаптироваться для достижения своей цели. В настоящее время наиболее плодотворные агенты обучения с подкреплением используют искусственные нейронные сети, что дает право называть их *алгоритмами глубокого обучения с подкреплением*.

В этой главе мы:

- Познакомимся с базовой теорией обучения с подкреплением в целом и с моделью, которую называют глубоким *Q*-обучением, в частности.
- С помощью библиотеки Keras построим глубокую сеть *Q*-обучения и обучим ее некоторым видеоиграм.
- Обсудим подходы к оптимизации агентов глубокого обучения с подкреплением.
- Познакомимся с семействами агентов глубокого обучения с подкреплением, не принадлежащими к классу глубокого *Q*-обучения.

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ГЛУБОКОГО ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ

Как рассказывалось в главе 4 (см. рис. 4.3), обучение с подкреплением — это парадигма машинного обучения:

- *Агент* выполняет *действия* в *окружении* (скажем, действие выполняется в некоторый момент времени t).

- Окружение возвращает агенту информацию двух типов:
  1. *Вознаграждение*: скалярное значение — количественная оценка действия, выполненного агентом в момент t. Например, это может быть 100 баллов награды за собранные вишни в видеоигре Pac-Man. Задача агента — максимизировать накапливаемые вознаграждения. Поэтому награды — это то, что *подкрепляет* продуктивное поведение, которое агент обнаруживает при определенных условиях.
  2. *Состояние*: как изменяется окружение в ответ на действия агента. В предстоящем временном шаге ($t + 1$) агент будет руководствоваться состоянием окружения, выбирая следующее действие.
- Два вышеупомянутых шага повторяются в цикле, пока не будет достигнуто некоторое конечное состояние. Оно может определяться достижением максимально возможной величины вознаграждения, определенного результата (например, автомобиль с автоматическим управлением прибыл в запрограммированный пункт назначения), исчерпанием отведенного времени, выполнением максимального количества разрешенных ходов или гибели агента в игре.

К классу задач обучения с подкреплением можно отнести все задачи, решаемые принятием последовательности решений. В главе 4 мы обсудили ряд конкретных примеров, в том числе:

- видеоигры для Atari, такие как Pac-Man, Pong и Breakout;
- автоматические транспортные средства, например беспилотные автомобили и летательные аппараты;
- настольные игры, такие как Го, шахматы и Сёги;
- управление роботизированными манипуляторами, например, извлекающими гвозди с помощью гвоздодера.

ИГРА CART-POLE

В этой главе мы используем OpenAI Gym — популярную библиотеку окружений для обучения с подкреплением (см. примеры на рис. 4.13), с помощью которой обучим агента игре Cart-Pole, представляющей классическую задачу в области теории управления. В игре Cart-Pole:

- Цель состоит в том, чтобы удерживать в вертикальном положении шест, установленный на тележке. Шест соединен с тележкой шарниром (изображен на рис. 13.1 кружком), который позволяет шесту вращаться влево или вправо[1].

[1] Фактический снимок экрана игры Cart-Pole показан на рис. 4.13, а.

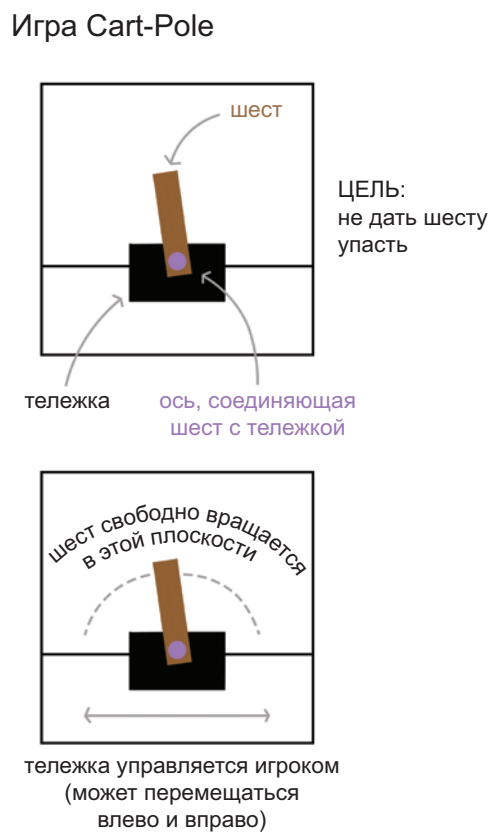


Рис. 13.1. Цель игры Cart-Pole — как можно дольше удерживать в вертикальном положении коричневый шест на черной тележке. Игрок (человек или компьютер) может перемещать тележку влево или вправо вдоль черной линии. Шест свободно вращается вокруг оси, которую создает фиолетовый штифт

- Тележка может перемещаться только влево или вправо. В любой данный *момент времени* тележка *должна* двигаться влево или вправо; она не может оставаться неподвижной.
- В начале каждого эпизода игры тележка находится в произвольной точке около центра экрана, и шест наклонен в ту или иную сторону на случайный угол, близкий к вертикали.
- Как показано на рис. 13.2, эпизод заканчивается, когда наступает одно из двух событий:
  - шест теряет равновесие, то есть слишком сильно отклоняется от вертикали;
  - тележка касается левой или правой границы экрана.

- В версии игры, о которой мы будем говорить в этой главе, максимальное число шагов в эпизоде равно 200. То есть если эпизод не закончится раньше (из-за потери баланса шеста или выхода за границы экрана), то он закончится через 200 шагов.
- За каждый шаг, пока длится эпизод, дается одно очко, поэтому максимально возможное вознаграждение составляет 200 очков.

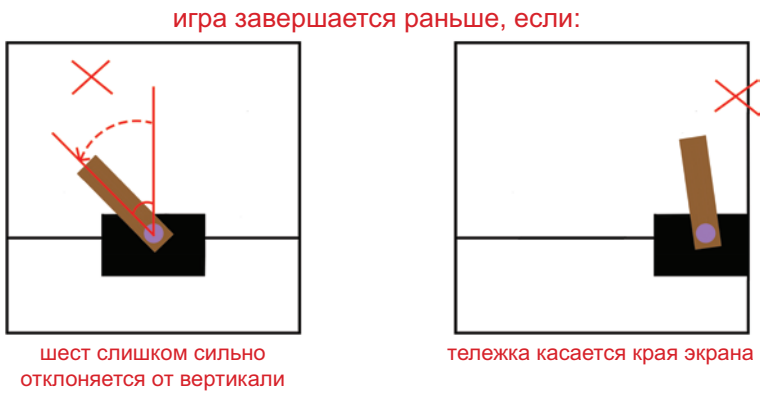


Рис. 13.2. Игра Cart-Pole завершается раньше, если шест слишком сильно отклоняется от вертикали или тележка касается края экрана

Игра Cart-Pole — популярная вводная задача для обучения с подкреплением, потому что она очень проста. В машине с автоматическим управлением существует практически бесконечное число возможных состояний окружения: когда она движется по дороге, ее многочисленные датчики — камеры, радар, лидар[1], акселерометры, микрофоны и т. д. — транслируют огромный объем информации о состоянии мира вокруг автомобиля, порядка гигабайта в секунду[2]. В игре Cart-Pole, напротив, есть всего четыре параметра, характеризующих состояние:

1. Координата тележки на горизонтальной оси.
2. Скорость тележки.
3. Угол наклона шеста.
4. Угловая скорость шеста.

Кроме того, автопилот, управляющий автомобилем, может производить ряд довольно тонких действий, таких как ускорение, торможение и поворот руля вправо или влево. В игре Cart-Pole в любой заданный момент времени t может выполняться ровно одно действие из двух возможных: движение влево или вправо.

[1] Действует по тому же принципу, что и радар, но вместо звука использует лазер.
[2] *bit.ly/GBpersec*

МАРКОВСКИЙ ПРОЦЕСС ПРИНЯТИЯ РЕШЕНИЙ

Задачи обучения с подкреплением на математическом языке можно определить как *марковский процесс принятия решений* (Markov Decision Process, MDP). Такие процессы обладают так называемым *марковским свойством* — предполагается, что текущий временной шаг содержит всю необходимую информацию о состоянии окружения, созданного предыдущими шагами. В отношении игры Cart-Pole это означает, что наш агент будет выбирать направление движения вправо или влево для заданного временного шага t, учитывая только состояние тележки (например, ее местоположение) и шеста (например, угол наклона) в этот конкретный шаг t.[1]

Как показано на рис. 13.3, марковский процесс принятия решений определяется пятью компонентами:

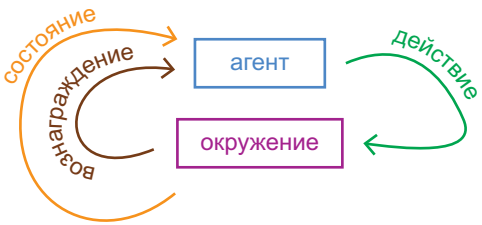
1. S — множество всех возможных *состояний*. В соответствии с соглашениями, принятыми в теории множеств, каждое отдельное возможное состояние (то есть конкретная комбинация положения и скорости тележки, угла наклона и угловой скорости шеста) представляется строчной буквой s. Даже если рассматривать относительно простую игру Cart-Pole, количество возможных комбинаций четырех параметров, описывающих состояние, огромно. Например, вот пара грубых примеров: тележка может медленно перемещаться вправо при вертикальном положении шеста или быстро перемещаться влево, при этом шест может с большой скоростью наклоняться вправо, по часовой стрелке.

2. A — это множество всех возможных *действий*. В игре Cart-Pole это множество содержит только два элемента (*влево* и *вправо*); в других окружениях их гораздо больше. Каждое отдельное возможное действие обозначается как a.

3. R — это распределение *вознаграждения* с учетом *пары состояние—действие* — определенного состояния в паре с определенным действием, обозначаемой как (s, a). Под словом «распределение» подразумевается распределение вероятностей: одна и та же пара состояние—действие (s, a) может случайным образом приводить к разным вознаграждениям r в разных случаях[2]. Детали распределения вознаграждения R — форма, включая его среднее значение и дисперсию, скрыты от агента, но могут быть опреде-

[1] Наличие марковского свойства предполагается во многих финансово-торговых стратегиях. Например, торговая стратегия может учитывать цену всех акций на бирже в конце данного торгового дня, но не учитывать данные за любой предыдущий день.

[2] Это верно в отношении обучения с подкреплением в целом, но игра Cart-Pole является относительно простым и полностью детерминированным окружением. В Cart-Pole одна и та же пара состояние—действие (s, a) всегда будет приносить одно и то же вознаграждение. Для иллюстрации принципов обучения с подкреплением в целом мы приводим в этом разделе примеры, которые предполагают меньшую детерминированность игры Cart-Pole, чем есть на самом деле.



Марковский процесс принятия решений

S: все возможные состояния

A: все возможные действия

R: распределение вознаграждений с учетом (s, a)

$\mathbb{P}$: вероятность перехода в s_{t+1} с учетом (s, a)

γ: коэффициент дисконтирования

Рис. 13.3. Цикл обучения с подкреплением (вверху; видоизмененная версия рис. 4.3, приводится здесь снова для удобства) можно рассматривать как марковский процесс принятия решений, который определяется пятью компонентами: S, A, R, $\mathbb{P}$ и γ (внизу)

лены выполнением действий в окружении. Например, на рис. 13.1 можно видеть, что тележка находится в центре экрана, а шест слегка наклонен влево[1]. Предполагается, что действие перемещения влево в этом состоянии s в среднем будет давать большее вознаграждение r, чем перемещение вправо в этом же состоянии: перемещение влево в этом состоянии s должно привести к уменьшению угла отклонения шеста от вертикали и увеличению числа шагов, в которых шест удерживался в равновесии, что, в свою очередь, ведет к увеличению вознаграждения r. Движение вправо в этом состоянии s, напротив, увеличит вероятность падения шеста, а значит, и более раннего завершения игры и уменьшения вознаграждения r.

4. $\mathbb{P}$ так же, как R, является распределением вероятности. В данном случае — вероятность следующего состояния (то есть s_{t+1}), заданного конкретной парой состояние—действие (s, a) в текущий момент времени t. Как и R, распределение $\mathbb{P}$ скрыто от агента, но его аспекты точно так же можно определить с помощью действий в окружении. Например, в игре Cart-Pole агент легко сможет узнать, что действие *влево* напрямую соответствует движению тележки влево[2]. Более сложные отношения — например, что действие

[1] Для простоты не будем учитывать в этом примере скорость тележки и угловую скорость падения шеста, потому что эти аспекты состояния нельзя вывести из статического изображения.

[2] Так же как все другие искусственные нейронные сети в этой книге, сети внутри агентов глубокого обучения с подкреплением инициализируются случайными начальными

влево в состоянии, изображенном на рис. 13.1, способствует уменьшению отклонения шеста от вертикали в следующем состоянии s_{t+1} — выучить будет труднее, и для этого потребуется больше времени.

5. γ (гамма) — это гиперпараметр, который называют *коэффициентом дисконтирования* (или *коэффициентом затухания*). Чтобы объяснить его назначение, отвлечемся от игры в Cart-Pole и вернемся к Pac-Man. Персонаж Pac-Man исследует двумерный лабиринт и получает призовые очки за сбор фруктов, но погибает, если столкнется с одним из призраков, преследующих его. Как показано на рис. 13.4, когда агент оценивает величину предполагаемого вознаграждения, он должен оценить, не является ли вознаграждение, которое можно получить немедленно (скажем, 100 очков за подобранную вишню, которая находится на расстоянии всего одного пиксела от Pac-Man), выше, чем эквивалентное вознаграждение, требующее большего количества шагов для достижения (100 очков за вишню, находящуюся на расстоянии 20 пикселов). Немедленное вознаграждение более ценно, чем отдаленное, потому что нельзя рассчитывать на отдаленное вознаграждение: на пути у Pac-Man может оказаться призрак или какая-то другая опасность[1,2]. Если установить $\gamma = 0.9$, тогда вишня, находящаяся на расстоянии одного шага, будет стоить 90 очков[3], а вишня, находящаяся на расстоянии 20 шагов, — всего 12 очков[4].

ОПТИМАЛЬНАЯ СТРАТЕГИЯ

Конечная цель марковского процесса принятия решений (MDP) состоит в том, чтобы найти функцию, которая позволит агенту выполнить соответствующее действие a (из множества всех возможных действий A), когда он встретит какое-либо конкретное состояние s из множества всех возможных состояний S.

параметрами. То есть перед любым обучением (например, посредством игры в Cart-Pole) агент ничего не знает даже о самых простых отношениях между некоторой парой действие—состояние (s, a) и следующим состоянием s_{t+1}. Например, человеку, играющему в игру Cart-Pole, может быть интуитивно понятно, что действие *влево* должно заставить тележку двигаться влево, но для случайно инициализированной нейронной сети *ничто* не является «интуитивным» или «очевидным», поэтому все отношения должны выявляться в процессе игры.

[1] Коэффициент дисконтирования γ можно сравнить с расчетами *дисконтированных денежных потоков*, которые широко используются в бухгалтерском учете: предполагаемый доход через год имеет меньшую ценность, чем доход, ожидаемый сегодня.

[2] Далее в этой главе мы введем понятия функции ценности (V) и функции Q-ценности (Q). Обе функции, V и Q, используют коэффициент γ, потому что он не дает им стать неограниченными (и, следовательно, вычислительно невозможными) в играх с бесконечным числом возможных временных шагов.

[3] $100 \times \gamma t = 100 \times 0.91 = 90$.

[4] $100 \times \gamma t = 100 \times 0.920 = 12.16$.

Другими словами, нам нужно, чтобы наш агент определил функцию, которая позволит ему *отображать S* в *A*. Как показано на рис. 13.5, такая функция обозначается как π и называется *функцией стратегии*.

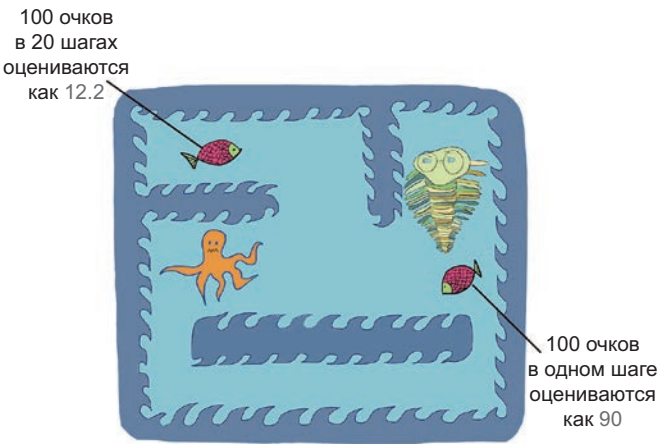


Рис. 13.4. При использовании коэффициента дисконтирования γ ценность более отдаленного вознаграждения в марковском процессе принятия решений ниже ценности вознаграждения, достижимого быстрее. Если для иллюстрации этой идеи использовать игру Pac-Man (зеленый трилобит представляет здесь мистера Pac-Man), тогда для $\gamma = 0.9$ вишня (или рыба), находящаяся всего в одном шаге, оценивается в 90 очков, а вишня (рыба) в 20 шагах оценивается в 12.2 очка. Так же как в игре Pac-Man, по лабиринту бродит осьминог и надеется убить бедного трилобита. Вот почему немедленно достижимые награды более ценны, чем отдаленные: у нас больше шансов быть убитым, прежде чем мы доберемся до рыбы, которая находится дальше

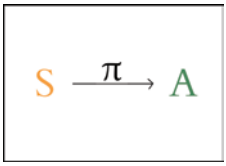


Рис. 13.5. Функция стратегии π позволяет агенту отображать любое состояние *s* (из множества всех возможных состояний *S*) в действие *a* из множества всех возможных действий *A*

Идея высокоуровневой функции стратегии π заключается в следующем: независимо от конкретных обстоятельств, какой *стратегии* должен придерживаться агент, чтобы максимизировать вознаграждение? Вот более конкретное определение идеи максимизации вознаграждения:

$$J(\pi^*) = \max_\pi J(\pi) = \max_\pi \mathbb{E}\left[\sum_{t>0} \gamma^t r_t\right]. \qquad (13.1)$$

В этом уравнении:

- $J(\pi)$ — называется *целевой функцией*. К этой функции можно применить методы машинного обучения, чтобы максимизировать вознаграждение[1].
- π представляет *любую* функцию стратегии, отображающую S в A.
- π^* представляет конкретную *оптимальную* стратегию (из всех возможных стратегий π) для отображения S в A. То есть π^* — это функция, которая для любого состояния s возвращает действие a, ведущее к получению агентом *максимально* возможного *дисконтированного вознаграждения*.
- *Ожидаемое дисконтированное вознаграждение* определяется выражением $\mathbb{E}\left[\sum_{t>0}\gamma^t r_t\right]$, где $\mathbb{E}$ обозначает *ожидание* (expectation), а $\sum_{t>0}\gamma^t r_t$ — *дисконтированное вознаграждение*.
- Чтобы вычислить дисконтированное вознаграждение $\sum_{t>0}\gamma^t r_t$ для всех временных шагов ($t > 0$), необходимо:
  - умножить вознаграждение, которое можно получить на любом шаге (r_t), на коэффициент дисконтирования этого временного шага (γ^t);
  - суммировать отдельные дисконтированные вознаграждения ($\gamma^t r_t$).

БАЗОВАЯ ТЕОРИЯ СЕТЕЙ ГЛУБОКОГО *Q*-ОБУЧЕНИЯ

В предыдущем разделе мы определили обучение с подкреплением как марковский процесс принятия решений. В конце мы указали, что было бы желательно, чтобы в ходе этого процесса наш агент, сталкиваясь с любым заданным состоянием s в любой заданный момент времени t, следовал некоторой оптимальной стратегии π^*, которая позволила бы ему выбрать действие a, максимизирующее дисконтированное вознаграждение. Проблема заключается в том, что даже при решении довольно простой задачи обучения с подкреплением, такой как игра в Cart-Pole, трудно (более или менее эффективно) рассчитать максимум совокупного дисконтированного вознаграждения $\max\left(\sum_{t>0}\gamma^t r_t\right)$. Из-за слишком

[1] Функции стоимости (или функции потерь), упоминавшиеся на протяжении всей книги, являются примерами целевых функций. В отличие от функции затрат, которая возвращает некоторое значение C стоимости, целевая функция $J(\pi)$ возвращает некоторое значение вознаграждения r. Используя функции стоимости, мы преследуем цель *минимизировать* стоимость, поэтому применяем градиентный *спуск* (как показано на рис. 8.2, где изображен трилобит, спускающийся вниз по склону). При использовании функции $J(\pi)$ наша цель, напротив, — *максимизировать* вознаграждение, поэтому мы технически применяем метод градиентного *восхождения* (вернитесь к рис. 8.2, подключите воображение и представьте, что трилобит взбирается на вершину горы), хотя математически это все тот же градиентный спуск.

большого числа возможных будущих состояний S и действий A, которые могут быть предприняты в этих состояниях, существует слишком много возможных вариантов развития будущего, чтобы учесть их все. Поэтому мы опишем подход *Q-обучения* к оценке оптимальности действия a в данной ситуации, уменьшающий вычислительную сложность.

ФУНКЦИИ ЦЕННОСТИ

Историю развития Q-обучения легче всего описать, начав с объяснения *функций ценности*. Функция ценности определяется как $V^{\pi}(s)$. Она позволяет получить представление о *ценности* данного состояния s, если наш агент последует стратегии π, начиная с этого состояния.

Как простой пример рассмотрим еще раз состояние s, изображенное на рис. 13.1[1]. Если предположить, что наш агент уже имеет некоторую разумную стратегию π для уравновешивания шеста, тогда совокупное дисконтированное вознаграждение, которое ожидается получить в этом состоянии, вероятно, довольно велико, потому что шест близок к вертикали. Ценность $V^{\pi}(s)$ этого конкретного состояния s высока.

С другой стороны, если представить состояние s_h, когда шест отклонен от вертикали на очень большой угол, его ценность — $V^{\pi}(s_h)$ — будет намного ниже, потому что агент уже потерял контроль над положением шеста и данный эпизод игры, вероятно, завершится в течение ближайших нескольких шагов.

ФУНКЦИИ *Q*-ЦЕННОСТИ

Функция Q-ценности[2] основана на функции ценности и учитывает не только состояние, но также полезность конкретного действия в данном состоянии, то есть перефразируя нашего старого друга, пары состояние—действие, обозначаемой как (s, a). Таким образом, если функция ценности определяется как $V^{\pi}(s)$, то функция Q-ценности определяется как $Q^{\pi}(s, a)$.

Вернемся снова к рис. 13.1. Объединение действия *влево* (назовем его a_L) с текущим состоянием s и дальнейшее следование стратегии балансировки шеста π должны принести высокое совокупное дисконтированное вознаграждение. То есть Q-ценность этой пары состояние—действие (s, a_L) высока.

Для сравнения рассмотрим выбор действия *вправо* (назовем его a_R) с состоянием s на рис. 13.1 с последующим следованием стратегии балансировки ше-

---

[1] Как мы уже делали выше в этой главе, рассмотрим только положение тележки и шеста, потому что, опираясь на эту статическую картинку, мы не можем строить предположений о скорости тележки или угловой скорости шеста.

[2] «Q» в названии Q-ценность означает *quality* (качество), но вы редко будете слышать, чтобы кто-то называл эти функции функциями качества—ценности.

ста. Несмотря на то что выбор этого действия может не оказаться вопиющей ошибкой, накопленное дисконтированное вознаграждение почти наверняка окажется несколько ниже, чем при выполнении действия *влево*. В этом состоянии *s* действие *влево* должно привести шест в положение ближе к вертикали (что упростит сохранение баланса шеста в будущем), тогда как действие *вправо* наклонит шест ближе к горизонтали, а значит, осложнит сохранение баланса и увеличит вероятность раннего завершения эпизода. Проще говоря, мы ожидаем, что *Q*-ценность пары (s, a_L) будет выше, чем *Q*-ценность пары (s, a_R).

ОЦЕНКА ОПТИМАЛЬНОЙ *Q*-ЦЕННОСТИ

Когда агент сталкивается с некоторым состоянием *s*, было бы желательно иметь возможность рассчитать *оптимальную Q-ценность*, обозначаемую как $Q^*(s, a)$. Можно было бы рассмотреть все возможные действия и выбрать в качестве наилучшего действие с наибольшей *Q*-ценностью — наибольшим совокупным дисконтированным вознаграждением.

Точно так же, как с вычислительной точки зрения трудно вычислить оптимальную стратегию π^* (уравнение 13.1) даже в относительно простых задачах обучения с подкреплением, трудно вычислить и оптимальную *Q*-ценность, $Q^*(s, a)$. Однако, используя подход глубокого *Q*-обучения (представленный в главе 4; см. рис. 4.5), можно создать искусственную нейронную сеть для *оценки* оптимальной *Q*-ценности. Такие сети глубокого *Q*-обучения (*Deep Q-Learning Network*, *DQN*) опираются на следующее уравнение:

$$Q^*(s, a) \approx Q(s, a; \theta). \tag{13.2}$$

В этом уравнении:

- Оптимальная *Q*-ценность ($Q^*(s, a)$) является *приближенной* оценкой.
- Функция аппроксимации *Q*-ценности, кроме состояния *s* и действия *a*, включает также параметры модели нейронной сети (обозначаются греческой буквой тета, θ). Это обычные веса и смещения искусственных нейронов, с которыми мы постоянно сталкиваемся, начиная с 6 главы.

Для более глубокого знакомства с теорией обучения с подкреплением, в том числе с глубокими сетями *Q*-обучения, мы рекомендуем недавнее издание книги Ричарда Саттона, см. рис. 13.6, и Эндрю Барто (Andrew Barto) «Reinforcement Learning: An Introduction»[1], которое доступно бесплатно по адресу *bit.ly/SuttonBarto*.

[1] *Sutton R. & Barto A.* (2018). Reinforcement Learning: An Introduction (2nd ed.). Cambridge, MA: MIT Press.

В контексте игры *Cart-Pole* агент *DQN*, опирающийся на уравнение 13.2, может, столкнувшись с определенным состоянием *s*, вычислить, даст ли действие *a* (*влево* или *вправо*) в данном состоянии увеличение прогнозируемого совокупного дисконтированного вознаграждения. Если, скажем, прогнозируется, что действие *влево* даст большее совокупное дисконтированное вознаграждение, значит, именно это действие следует предпринять. В следующем разделе мы реализуем с помощью библиотеки *Keras* агента *DQN*, включающего полносвязанную нейронную сеть, чтобы показать, как это делается.

Рис. 13.6. Ричард Саттон, крупнейшая звезда в области обучения с подкреплением, долгое время был профессором информатики в Университете Альберты. В последнее время является также выдающимся научным сотрудником в Google DeepMind

ОПРЕДЕЛЕНИЕ АГЕНТА DQN

Наш код определяет агента DQN, который учится действовать в окружении — в данном случае в игре Cart-Pole — из библиотеки OpenAI Gym. Вы найдете его в блокноте Jupyter *cartpole_dqn.ipynb*[1]. Вот его зависимости:

```
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
import os
```

Самым значительным новшеством в этом списке является модуль `gym`, представляющий саму библиотеку Open AI Gym. Как обычно, мы подробно обсудим все зависимости по мере их применения.

Гиперпараметры, настраиваемые в верхней части блокнота, приводятся в листинге 13.1.

Листинг 13.1. Гиперпараметры агента DQN, обучающегося игре в Cart-Pole

```
env = gym.make('CartPole-v0')
state_size = env.observation_space.shape[0]
```

[1] Наш агент DQN основан на агенте Кеона Кима (Keon Kim), который доступен в его репозитории GitHub по адресу bit.ly/keonDQN.

```
action_size = env.action_space.n
batch_size = 32
n_episodes = 1000
output_dir = 'model_output/cartpole/'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

Рассмотрим этот код подробнее, строку за строкой:

- Вызовом метода `make()` из библиотеки Open AI Gym производится выбор окружения для обучения нашего агента. В данном случае выбирается нулевая версия (`v0`) игры Cart-Pole и присваивается переменной `env`. Вы можете выбрать альтернативное окружение Open AI Gym из числа представленных на рис. 4.13.
- Из окружения извлекаются два параметра:
  1. `state_size`: число типов информации о состоянии, которое для игры Cart-Pole равно 4 (напомню, что это положение тележки, скорость тележки, угол наклона шеста и угловая скорость шеста).
  2. `action_size`: число возможных действий, которое для игры Cart-Pole равно 2 (*влево* и *вправо*).
- Устанавливается размер пакета для обучения нашей нейронной сети: `32`.
- Устанавливается число эпизодов (раундов игры) равным `1000`. Как вы увидите ниже, это почти правильное число эпизодов, необходимых агенту, в течение которых он неизменно будет превосходить себя в игре Cart-Pole. В более сложных окружениях может потребоваться увеличить этот гиперпараметр, чтобы позволить агенту обучаться на протяжении большего числа раундов игры.
- Определяется уникальное имя каталога (`'model_output/cartpole/'`), куда регулярно будут сохраняться параметры нейронной сети. Если каталог не существует, мы создаем его с помощью `os.makedirs()`.

В листинге 13.2 представлен довольно большой фрагмент кода, в котором определяется класс `DQNAgent` агента DQN.

Листинг 13.2. Агент глубокого *Q*-обучения

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95
        self.epsilon = 1.0
        self.epsilon_decay = 0.995
        self.epsilon_min = 0.01
```

```
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(32, activation='relu',
                  input_dim=self.state_size))
        model.add(Dense(32, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse',
                      optimizer=Adam(lr=self.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action,
                            reward, next_state, done))

    def train(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward # если достигнут конец
            if not done:
                target = (reward +
                          self.gamma *
                          np.amax(self.model.predict(next_state)[0]))
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])

    def save(self, name):
        self.model.save_weights(name)

    def load(self, name):
        self.model.load_weights(name)
```

ИНИЦИАЛИЗАЦИЯ ПАРАМЕТРОВ

Листинг 13.2 начинается с инициализации атрибутов класса:

- `state_size` и `action_size` зависят от конкретного окружения; для игры Cart-Pole они инициализируются значениями `4` и `2` соответственно, как отмечалось выше.
- `memory` — этот параметр предназначен для хранения *воспоминаний*, которые впоследствии можно *воспроизвести* для обучения нейронной сети агента

DQN. Воспоминания хранятся в виде элементов структуры данных *deque* (произносится как «дек»), подобной списку, которая хранит только `maxlen = 2000` последних воспоминаний. То есть при попытке добавить в этот список 2001-й элемент его первый элемент удаляется, благодаря чему мы всегда имеем список, содержащий не более 2000 элементов.

- `gamma` — коэффициент дисконтирования (или скорость затухания) γ, который мы обсудили выше в этой главе (см. рис. 13.4). Этот гиперпараметр уменьшает величину вознаграждения в зависимости от его удаленности во времени. Обычно эффективнее выбирать значения γ, близкие к 1 (например, 0.9, 0.95, 0.98 и 0.99). Чем ближе этот коэффициент к 1, тем меньше обесценивается отдаленное вознаграждение[1]. Настройка гиперпараметров моделей обучения с подкреплением, таких как γ, может оказаться сложной задачей; в конце этой главы мы обсудим инструмент под названием SLM Lab, помогающий решать ее более эффективно.

- `epsilon` — обозначается греческой буквой ϵ — это еще один гиперпараметр обучения с подкреплением, который называют *долей исследовательских действий*. Он определяет долю от общего числа действий, которые агент будет выбирать случайно (что позволит ему *исследовать* влияние этих действий на следующее состояние s_{t+1} и вознаграждение r, возвращаемое окружением), относительно действий, выбираемых с *использованием* существующих «знаний», накопленных нейронной сетью в процессе игры. До начала первого эпизода агент не имеет опыта в игре, поэтому на практике принято начинать со стопроцентной доли исследовательских действий; вот почему мы установили `epsilon = 1.0`.

- По мере приобретения игрового опыта агентом мы очень медленно *снижаем* долю исследовательских действий, чтобы он мог использовать полученную информацию (в надежде, что это позволит ему получить большее вознаграждение, как показано на рис. 13.7). То есть в конце каждого эпизода мы умножаем долю исследовательских действий ϵ на `epsilon_decay`. Типичными значениями для этого гиперпараметра являются: 0.990, 0.995 и 0.999[2].

- `epsilon_min` — это минимальное значение, до которого может уменьшиться доля исследовательских действий ϵ. Обычно этому гиперпараметру присваивается значение, близкое к нулю, такое как 0.001, 0.01 или 0.02. Мы выбрали значение `0.01`, то есть после того, как ϵ уменьшится до 0.01 (в нашем случае это произойдет в 911-м эпизоде), агент будет выделять для исследования только 1% действий, а остальные 99% будут выбираться на основе накопленного игрового опыта.

---

[1] Если установить γ = 1 (что не рекомендуется), вознаграждения не будут обесцениваться с течением времени.

[2] По аналогии с настройкой γ = 1 настройка `epsilon_decay = 1` означает, что ϵ не будет снижаться, то есть исследования будут протекать с постоянной скоростью. Однако это весьма необычный выбор для данного гиперпараметра.

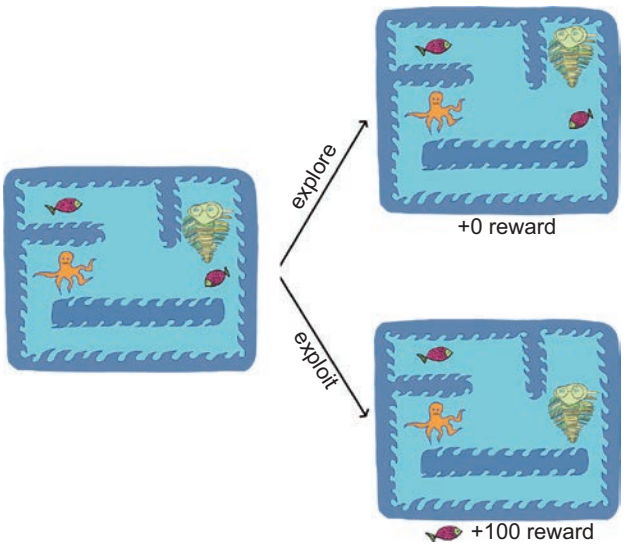


Рис. 13.7. Как и на рис. 13.4, здесь для иллюстрации идеи обучения с подкреплением мы используем окружение Pac-Man (с зеленым трилобитом, представляющим агента DQN вместо персонажа Pac-Man). В данном случае сопоставляется исследовательское поведение с поведением, основанном на накопленных знаниях. Чем выше гиперпараметр ϵ (эпсилон) в данном эпизоде, тем выше вероятность, что агент будет находиться в исследовательском режиме, в котором он выполняет чисто случайные действия: агент может случайно двинуться в направлении, противоположном пути к рыбе, выбор которого обеспечил бы немедленное вознаграждение в 100 очков. Альтернативой исследовательскому режиму является режим действий, основанных на накопленных знаниях. Если предположить, что параметры нейронной сети агента DQN уже были скорректированы в ходе предыдущих эпизодов, в режиме действий, основанных на накопленных знаниях, агент должен стремиться получить немедленно доступное вознаграждение

- `learning_rate` — это тот же гиперпараметр стохастического градиентного спуска, который мы рассмотрели в главе 8.
- В завершение вызывается метод `_build_model()` — символ подчеркивания в его имени подсказывает, что это приватный (закрытый) метод, то есть его рекомендуется использовать только внутри экземпляров класса `DQNAgent`.

СОЗДАНИЕ МОДЕЛИ НЕЙРОННОЙ СЕТИ АГЕНТА

Метод `_build_model()` из листинга 13.2 предназначен для создания и компиляции нейронной сети, которая отображает состояние среды s в Q-ценность для каждого доступного действия a. После обучения в игровом окружении агент сможет использовать предсказанные значения Q-ценности для выбора конкретного действия, которое он должен предпринять, опираясь на конкретное состояние окружения. В этом методе нет ничего, чего вы не видели выше в этой книге. Здесь мы:

- Создаем последовательную модель.
- Добавляем в нее следующие слои:
  - Первый скрытый слой — полносвязанный, содержащий 32 нейрона с функцией активации ReLU. Используя аргумент `input_dim`, мы указываем форму входного слоя сети — размерность информации о состоянии окружения *s*. В окружении Cart-Pole это массив с длиной 4, элементы которого представляют положение тележки, ее скорость, угол наклона шеста и его угловую скорость[1].
  - Второй скрытый слой — тоже полносвязанный с 32 нейронами ReLU. Как упоминалось выше, мы рассмотрим задачу выбора гиперпараметров, в том числе и архитектуры конкретной модели, когда будем обсуждать инструмент SLM Lab далее в этой главе.
  - Выходной слой имеет размерность, соответствующую числу возможных действий[2]. В данном случае размерность соответствует количеству возможных действий в игре Cart-Pole: это массив с длиной 2, один элемент которого соответствует действию *влево*, а другой — *вправо*. Так же как в модели регрессии (см. листинг 9.8), значения *z* выводятся агентом DQN непосредственно из нейронной сети, без преобразования в вероятности между 0 и 1. Для этого мы использовали функцию активации `linear` вместо `sigmoid` или `softmax`.
- Как отмечалось, когда мы компилировали регрессионную модель (листинг 9.9), среднеквадратичная ошибка прекрасно подходит на роль функции стоимости, когда в выходном слое используется линейная активация, поэтому методу `compile()` мы передаем аргумент `loss` со значением `mse`. При определении оптимизатора возвращаемся к нашему обычному выбору `Adam`.

ЗАПОМИНАНИЕ ИГРОВОГО ПРОЦЕССА

В каждый момент времени *t*, то есть в любой итерации цикла обучения с подкреплением (см. рис. 13.3), вызывается метод `remember()` агента DQN, добав-

[1] В других окружениях информация о состоянии может иметь гораздо более сложный вид. Например, в окружении игры Pac-Man состояние *s* состоит из пикселов на экране, то есть здесь мы имеем дело с двух- или трехмерным вводом (для монохромного или полноцветного изображения соответственно). В такой ситуации на роль первого скрытого слоя лучше выбрать сверточный слой, такой как `Conv2D` (см. главу 10).

[2] Все предыдущие модели в этой книге, возвращающие только два результата (как в главах 11 и 12), использовали сигмоидный нейрон. Здесь для каждого результата мы создали отдельные нейроны, потому что хотели, чтобы наш код можно было использовать для обучения не только в игре Cart-Pole. В Cart-Pole есть только два действия, но во многих других окружениях их больше двух.

ляющий информацию в конец списка `memory`. Каждый элемент в этом списке содержит пять типов информации о временном шаге t:

1. Состояние s_t (`state`), встретившееся агенту.
2. Действие a_t (`action`), предпринятое агентом.
3. Вознаграждение r_t (`reward`), которое вернуло окружение в ответ на действие.
4. Следующее состояние s_{t+1} (`next_state`), которое вернуло окружение.
5. Логический флаг `done`, который получает значение `True`, если достигнут последний временной шаг в эпизоде, и `False` в ином случае.

ОБУЧЕНИЕ ПОСРЕДСТВОМ ВОСПРОИЗВЕДЕНИЯ ВОСПОМИНАНИЙ

Модель нейронной сети агента DQN обучается путем *воспроизведения воспоминаний* об игровом процессе, как можно видеть в методе `train()` в листинге 13.2. Процесс начинается со случайного отбора данных в пакет с размером 32 (в соответствии с параметром агента `batch_size`) из списка `memory` (который вмещает до 2000 элементов). Выбор небольшого подмножества воспоминаний из гораздо большего набора, представляющего опыт агента, увеличивает эффективность обучения модели: если бы для обучения мы использовали, скажем, 32 самых последних воспоминания, многие состояния в этих воспоминаниях были бы очень похожими. Чтобы лучше понять, о чем речь, представьте временной шаг t, в котором тележка находится в каком-то конкретном месте, а шест занимает почти вертикальное положение. В смежные временные шаги (например, $t - 1$, $t + 1$, $t + 2$) тележка почти наверняка будет находиться почти в той же точке, и шест будет занимать почти то же вертикальное положение. При выборе из более широкого диапазона воспоминаний, отдаленных во времени друг от друга, модель получит в каждом цикле обучения более богатый опыт.

Отобрав пакет с 32 воспоминаниями, мы проводим обучение модели: если `done` равно `True`, то есть воспоминание соответствует последнему временному шагу в эпизоде, мы точно знаем, что максимально возможное вознаграждение на этом временном шаге равно `reward`. Поэтому можно просто установить целевое вознаграждение `target` равным `reward`.

В противном случае (то есть если `done` равно `False`) мы пытаемся оценить величину целевого вознаграждения `target` как максимальное дисконтированное вознаграждение. Для этого берем известное вознаграждение `reward` и добавляем к нему дисконтированную[1] максимальную Q-ценность. Возможные Q-ценности в будущем определяются путем передачи следующего (то есть *будущего*) состояния s_{t+1} в метод `predict()` модели. В контексте игры Cart-Pole этот метод возвращает два результата: один для действия *влево* и другой для действия

[1] То есть умноженную на `gamma`, коэффициент дисконтирования γ.

вправо. Наибольший из этих двух результатов (определяется с помощью функции `amax` из пакета NumPy) принимается как максимальная прогнозируемая *Q*-ценность в будущем.

Независимо от того, известно ли целевое значение `target` (потому что временной шаг был последним в эпизоде) или оно было оценено вычислением максимальной *Q*-ценности, далее в цикле `for`, в методе `train()`:

- Снова вызывается метод `predict()`, которому передается *текущее состояние* s_t. Как и прежде, в контексте игры Cart-Pole этот метод возвращает два результата: один для действия *влево* и один для действия *вправо*. Мы сохраняем эти два вывода в переменной `target_f`.
- Независимо от действия a_t (`action`), выполненного агентом из памяти, мы используем `target_f [0] [action] = target`, чтобы заменить результат `target_f` целевым вознаграждением `target`[1].
- Мы обучаем модель вызовом метода `fit()`.
  - На вход модели подается текущее состояние s_t (`state`), а на выходе получается значение `target_f`, включающее оценку максимального дисконтированного вознаграждения в будущем. Настраивая параметры модели (представленные членом θ в уравнении 13.2), мы улучшаем ее способность точно прогнозировать действие, которое, скорее всего, будет связано с максимизацией будущего вознаграждения в любом данном состоянии.
  - Во многих задачах обучения с подкреплением переменной `epochs` можно присвоить значение `1`. Вместо многократного обучения на существующем наборе обучающих данных можно поучаствовать в большем количестве эпизодов игры Cart-Pole (например) и сгенерировать столько новых обучающих данных, сколько понадобится.
  - Мы установили `verbose = 0`, потому что на данном этапе нам не нужна никакая информация о том, как протекает процесс обучения. Как будет показано ниже, вместо этого мы будем следить за изменением эффективности агентов в каждом эпизоде.

ВЫБОР ДЕЙСТВИЯ

Для выбора конкретного действия в данный момент времени *t* используется метод агента `act()`. Внутри этого метода используется функция `rand` из библиотеки NumPy, возвращающая случайное значение в диапазоне от 0 до 1,

[1] Это делается потому, что выяснить величину *Q*-ценности можно только на основе действий, фактически выполненных агентом: мы оценили цель `target` на основе следующего состояния s_{t+1} (`next_state`) и знаем только, что к состоянию s_{t+1} привело действие a_t (`action`), фактически выполненное агентом на шаге *t*. Мы не знаем, какое следующее состояние s_{t+1} вернуло бы окружение, если бы агент выполнил какое-то другое действие.

которое мы будем называть v. В сочетании с гиперпараметрами агента — `epsilon`, `epsilon_decay` и `epsilon_min` — значение v будет определять выбор между исследовательским действием и действием на основе опыта[1]:

- Если случайное значение v меньше или равно доле исследовательских действий ϵ, с помощью функции `randrange` выбирается случайное исследовательское действие. На начальных этапах обучения, когда значение ϵ еще достаточно высоко, большинство действий будет носить исследовательский характер. На более поздних этапах, по мере уменьшения ϵ (в соответствии с гиперпараметром `epsilon_decay`), агент будет выполнять все меньше и меньше исследовательских действий.
- Иначе, то есть если случайное значение v больше ϵ, выбирается действие на основе знаний, накопленных моделью. Чтобы использовать это знание, состояние s_t (`state`) передается в метод `predict()` модели, который возвращает результат активации[2] для каждого из возможных действий. Для выбора действия a_t с наибольшим значением активации используется функция `argmax` из библиотеки NumPy.

СОХРАНЕНИЕ И ЗАГРУЗКА ПАРАМЕТРОВ МОДЕЛИ

Наконец, однострочные методы `save()` и `load()` дают возможность сохранять и загружать параметры модели. В частности, в сложных окружениях эффективность агента может быть непостоянной: на длительных отрезках агент может показывать очень хорошую эффективность в данном окружении, а затем по какой-то причине полностью утратить свои способности. Так что целесообразно сохранять параметры модели через равные промежутки времени. Затем, если эффективность агента упадет на более поздних этапах, в модель можно загрузить высокоэффективные параметры, полученные на предыдущих.

ВЗАИМОДЕЙСТВИЕ С ОКРУЖЕНИЕМ ИЗ OPENAI GYM

Создав класс агента DQN, мы можем запустить экземпляр класса и сохранить его в переменной `agent`, как показано ниже:

```
agent = DQNAgent(state_size, action_size)
```

Код в листинге 13.3 реализует взаимодействие нашего агента с окружением из библиотеки OpenAI Gym, в данном случае — с окружением игры Cart-Pole.

---

[1] Порядок выбора исследовательских действий и действий на основе накопленного опыта был описан выше, когда обсуждалась инициализация параметров нашего класса `DQNAgent`, и дополнительно проиллюстрирован на рис. 13.7.

[2] Напомним, что здесь используется линейная активация, а это значит, что результат не является вероятностью — это дисконтированное вознаграждение за действие.

Листинг 13.3. Взаимодействие агента DQN с окружением из OpenAI Gym

```
for e in range(n_episodes):
    state = env.reset()
    state = np.reshape(state, [1, state_size])
    done = False
    time = 0
    while not done:
#         env.render()
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        reward = reward if not done else -10
        next_state = np.reshape(next_state, [1, state_size])
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        if done:
            print("episode: {}/{}, score: {}, e: {:.2}"
                  .format(e, n_episodes-1, time, agent.epsilon))
        time += 1
    if len(agent.memory) > batch_size:
        agent.train(batch_size)
    if e % 50 == 0:
        agent.save(output_dir + "weights_"
                   + '{:04d}'.format(e) + ".hdf5")
```

Как вы наверняка помните, мы установили гиперпараметр `n_episodes` равным `1000`. Листинг 13.3 содержит длинный цикл `for`, который обеспечивает участие агента в этих 1000 эпизодах игры. Текущий эпизод сохраняется в переменной `e` и включает следующее:

- Вызов `env.reset()` запускает эпизод со случайным начальным состоянием s_t. Для передачи состояния `state` в нейронную сеть в том виде, в котором модель ожидает получить его, используется метод `reshape`, преобразующий столбец в строку[1].
- Внутри цикла `for`, перебирающего тысячу эпизодов, находится цикл `while`, который выполняет шаги в данном эпизоде. Пока эпизод не завершится (то есть пока переменная `done` не получит значение `True`), на каждом шаге t (представленном переменной `time`) выполняются следующие действия.
  - Инструкция `env.render()` закомментирована, потому что, если запустить этот код внутри блокнота Jupyter в контейнере Docker, эта инструкция вызовет ошибку. Но если вы решите запустить код каким-то другим способом (например, в блокноте Jupyter *не* в контейнере Docker), можете попробовать раскомментировать эту инструкцию. Если ошибки не возникнет, должно появиться всплывающее окно, *отображающее* игровое поле. Это позволит вам в режиме реального времени наблюдать за действиями

[1] Ранее, в листинге 9.11, мы уже использовали эту операцию транспонирования по той же причине.

агента DQN, играющего в игру Cart-Pole, эпизод за эпизодом. Зрелище, конечно, захватывающее, но наличие или отсутствие этого окна никак не влияет на обучение агента!

- Состояние s_t (`state`) передается методу `act()` агента, который возвращает действие a_t (`action`) — `0` (*влево*) или `1` (*вправо*).
- Действие a_t передается методу окружения `step()`, который возвращает следующее состояние s_{t+1} (`next_state`), текущее вознаграждение r_t (`reward`) и логический флаг `done`.
- Если эпизод завершился (то есть переменная `done` получила значение `True`), переменной `reward` (вознаграждения) присваивается отрицательное значение (`-10`). Это вынуждает агента досрочно завершить эпизод при потере контроля над балансом шеста или перемещении тележки за пределы экрана. Если эпизод еще не завершился (переменная `done` получила значение `False`), значение переменной `reward` увеличивается на единицу за каждый дополнительный шаг в игре.
- Вызывается метод `reshape`, чтобы преобразовать `next_state` в строку, подобно тому, как в начале диапазона было преобразовано состояние `state`.
- Для сохранения всех аспектов текущего шага (состояния s_t, действия a_t, вознаграждения r_t, следующего состояния s_{t+1} и флага `done`) вызывается метод `remember()` агента.
- Для подготовки к следующей итерации, представляющей шаг $t + 1$, `next_state` переписывается в `state`.
- Если эпизод завершился, выводятся итоговые показатели, достигнутые в этом эпизоде (как показано на рис. 13.8 и 13.9).
- К счетчику шагов `time` прибавляется `1`.

- Если длина списка, представляющего память агента, превысила размер пакета, вызывается метод `train()` агента для корректировки параметров его нейронной сети воспроизведением воспоминаний об игровом процессе[1].
- Через каждые `50` эпизодов вызывается метод `save()` агента, чтобы сохранить параметры модели нейронной сети.

Как показано на рис. 13.8, в течение первых 10 эпизодов наш агент показывал весьма низкие результаты. Ему никак не удавалось выполнить больше 42 шагов в игре (заработать больше 41 очка). В ходе этих первых эпизодов доля исследовательских действий ϵ составляла 100%. К десятому эпизоду

[1] При желании вы можете перенести этот шаг обучения вверх, чтобы он оказался внутри цикла `while`. В этом случае каждый эпизод будет длиться *намного* дольше, потому что агент будет обучаться гораздо чаще, зато ему понадобится меньше эпизодов для обучения.

значение ϵ уменьшилось до 96%, то есть агент использовал накопленный опыт (см. рис. 13.7) для выбора действий примерно в 4% шагов. Однако на этом раннем этапе обучения большинство из таких действий на основе опыта фактически были случайными.

Как показано на рис. 13.9, к 991-му эпизоду наш агент освоил игру Cart-Pole. Во всех последних 10 эпизодах он набрал идеальный счет 199, достигая в каждом максимального числа шагов, равного 200. К 911-му эпизоду[1] доля исследовательских действий ϵ достигла своего минимума в 1%, поэтому во всех этих последних эпизодах примерно в 99% шагов агент выбирал действия, опираясь на полученный опыт. Судя по идеальным результатам, достигнутым в конце, можно сделать вывод, что нейронная сеть, управляющая выбором действий на основе опыта, хорошо обучилась игре в предыдущих эпизодах.

```
episode: 0/999, score: 19, e: 1.0
episode: 1/999, score: 14, e: 1.0
episode: 2/999, score: 37, e: 0.99
episode: 3/999, score: 11, e: 0.99
episode: 4/999, score: 35, e: 0.99
episode: 5/999, score: 41, e: 0.98
episode: 6/999, score: 18, e: 0.98
episode: 7/999, score: 10, e: 0.97
episode: 8/999, score: 9, e: 0.97
episode: 9/999, score: 24, e: 0.96
```

Рис. 13.8. Результаты, достигнутые агентом DQN в первых 10 эпизодах игры Cart-Pole. Результаты очень низкие (агенту удавалось выполнить в игре от 10 до 42 шагов), а доля исследовательских действий ϵ — высокая (100% в начале и 96% в 10 эпизоде)

```
episode: 990/999, score: 199, e: 0.01
episode: 991/999, score: 199, e: 0.01
episode: 992/999, score: 199, e: 0.01
episode: 993/999, score: 199, e: 0.01
episode: 994/999, score: 199, e: 0.01
episode: 995/999, score: 199, e: 0.01
episode: 996/999, score: 199, e: 0.01
episode: 997/999, score: 199, e: 0.01
episode: 998/999, score: 199, e: 0.01
episode: 999/999, score: 199, e: 0.01
```

Рис. 13.9. Результаты, достигнутые агентом DQN в последних 10 эпизодах игры Cart-Pole. Ему удалось выполнить максимальное (199) число шагов во всех 10 эпизодах. Доля исследовательских действий ϵ снизилась до минимума в 1%, поэтому в 99% случаев агент выбирает действия, опираясь на накопленный опыт

[1] Промежуточные результаты здесь не показаны, но вы можете увидеть их в нашем блокноте Jupyter *cartpole_dqn.ipynb*.

Как упоминалось выше в этой главе, агенты в глубоком обучении с подкреплением часто демонстрируют весьма странное поведение. Обучая своего агента DQN игре в Cart-Pole, вы можете обнаружить, что он показывает очень высокие результаты в некоторых более поздних эпизодах (достигая 200 шагов много эпизодов подряд, например, в 850-м или 900-м эпизоде), но затем резко теряет эффективность в последнем (1000-м). Столкнувшись с такой ситуацией, попробуйте использовать метод `load()` для восстановления параметров модели их более ранней и более эффективной фазы.

ОПТИМИЗАЦИЯ ГИПЕРПАРАМЕТРОВ С ПОМОЩЬЮ SLM LAB

Выше в этой главе мы залпом представили гиперпараметры и отметили, что позже познакомимся с инструментом под названием SLM Lab для настройки гиперпараметров[1]. Этот момент настал!

SLM Lab — это фреймворк глубокого обучения с подкреплением, разработанный Ва Лун Кенгом (Wah Loon Keng) и Лаурой Грейссер (Laura Graesser), инженерами-программистами из Калифорнии (первый работает в компании MZ, занимающейся разработкой мобильных игр, а второй входит в состав команды Google Brain). Фреймворк доступен по адресу github.com/kengz/SLM-Lab и имеет широкий спектр реализаций и функциональных возможностей, связанных с глубоким обучением с подкреплением:

- Позволяет использовать самые разные типы агентов глубокого обучения с подкреплением, в том числе DQN и другие (будут представлены далее в этой главе).
- Предлагает компоненты для конструирования агентов, что позволяет придумывать новые категории агентов глубокого обучения с подкреплением.
- Дает возможность обучать агентов в окружениях, входящих в состав различных библиотек, таких как OpenAI Gym и Unity (см. главу 4).
- Позволяет обучать агентов в нескольких окружениях одновременно. Например, один и тот же агент DQN может одновременно обучаться игре Cart-Pole из OpenAI Gym и игре балансировки мячей Ball2D из Unity.
- Позволяет сравнивать эффективность агента в разных окружениях.

Для нас особенно важно, что SLM Lab предлагает простой способ подбора различных гиперпараметров агента и оценки их влияния на эффективность агента в данном окружении. Рассмотрим, например, *график проведения эксперимента*

[1] Аббревиатура SLM расшифровывается как *Strange Loop Machine* (странная циклическая машина), причем понятие *strange loop* (странная петля) связано с идеей опыта человеческого сознания. *Hofstadter R.* (1979). Gödel, Escher, Bach. New York: Basic Books.

на рис. 13.10. В этом эксперименте агент DQN обучался игре в Cart-Pole в ряде *испытаний*. В каждом из них использовался агент с определенными гиперпараметрами, обученный на множестве эпизодов. Вот некоторые гиперпараметры, которые изменялись между испытаниями:

- Архитектура модели полносвязанной сети:
  - `[32]`: один скрытый слой с 32 нейронами;
  - `[64]`: также единственный скрытый слой, но уже с 64 нейронами;
  - `[32, 16]`: два скрытых слоя; первый с 32 нейронами и второй с 16 нейронами;
  - `[64, 32]`: также два скрытых слоя; первый с 64 нейронами и второй с 32 нейронами.
- Функция активации во всех скрытых слоях:
  - sigmoid;
  - tanh;
  - ReLU.
- Скорость обучения (η) изменялась от нуля до 0.2.
- *Продолжительность снижения* (*annealing*) доли исследовательских действий (ϵ) изменялась в диапазоне от 0 до 100[1].

SLM Lab предоставляет ряд метрик для оценки эффективности модели (часть из них можно видеть вдоль вертикальной оси на рис. 13.10):

- *Эффективность (strength)*: суммарное вознаграждение, накопленное агентом.
- *Скорость (speed)*: как быстро (то есть через сколько эпизодов) агент достиг максимальной эффективности.
- *Стабильность (stability)*: описывает способность агента сохранять эффективность в последующих эпизодах после достижения наивысшей точки.

---

[1] Продолжительность снижения (annealing — имитация отжига, охлаждения) является альтернативой коэффициента затухания ϵ и служит той же цели. Если для гиперпараметров `epsilon` и `epsilon_min` заданы фиксированные значения (скажем, 1.0 и 0.01 соответственно), в версии с гиперпараметром продолжительности снижения значение `epsilon_decay` будет корректироваться так, чтобы значение $\epsilon = 0.01$ было достигнуто в указанном эпизоде. Например, если продолжительность снижения выбрать равной `25`, то значение ϵ будет равномерно уменьшаться так, что достигнет целевого значения 0.01 к 25-му эпизоду. Если продолжительность снижения выбрать равной `50`, то значение ϵ будет равномерно уменьшаться так, что достигнет целевого значения 0.01 к 50-му эпизоду.

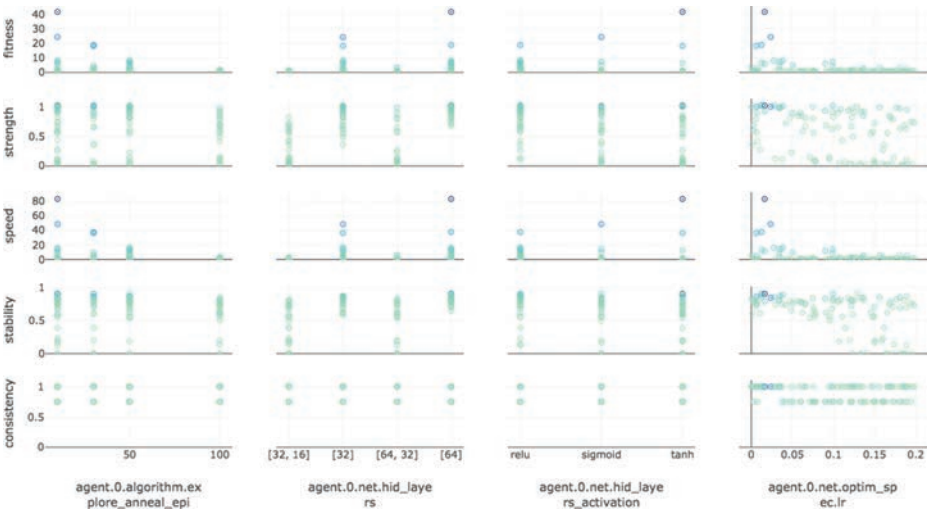


Рис. 13.10. Эксперимент, проведенный с SLM Lab, имел целью выяснить влияние различных гиперпараметров (например, архитектуры скрытых слоев, функции активации, скорости обучения) на эффективность агента DQN в среде Cart-Pole

- *Последовательность (consistency)*: описывает воспроизводимость результатов в разных испытаниях, когда агенты имели одинаковые настройки гиперпараметров.
- *Качество (fitness)*: сводная метрика, учитывающая все четыре, перечисленные выше. Исходя из анализа метрики качества в эксперименте, представленном на рис. 13.10, можно заключить, что для данного агента DQN, играющего в игру Cart-Pole, оптимальными являются следующие значения гиперпараметров:
  - нейронная сеть с единственным скрытым слоем, содержащим 64 нейрона; она превзошла аналогичную сеть с 32 нейронами;
  - функция активации tanh в скрытом слое;
  - низкая скорость обучения (η) ~0.02;
  - испытания, в которых продолжительность снижения доли исследовательских действий (ϵ) была равна 10, показали лучшие результаты, чем те, где продолжительность снижения была установлена равной 50 или 100.

Описание деталей работы SLM Lab выходит за рамки нашей книги, но для этого фреймворка имеется превосходная документация, доступная по адресу *kengz.gitbooks.io/slm-lab*[1].

[1] На момент написания этих строк фреймворк SLM Lab можно было без проблем установить только в системах на основе Unix, включая macOS.

ДРУГИЕ АГЕНТЫ, ОТЛИЧНЫЕ ОТ DQN

В мире глубокого обучения с подкреплением сети глубокого Q-обучения, подобные той, что мы создали выше, относительно просты. Агенты DQN не только просты (сравнительно), но и эффективно используют доступные обучающие выборки по сравнению со многими другими агентами глубокого обучения с подкреплением. Тем не менее агенты DQN имеют свои недостатки. Вот наиболее заметные из них:

1. Если в данном окружении возможное количество пар состояние—действие велико, то Q-функция может получиться чрезвычайно сложной, из-за чего становится трудно оценить оптимальную Q-ценность Q^*.
2. Даже в ситуациях, когда поиск Q^* осуществим с вычислительной точки зрения, агенты DQN плохо справляются с исследованием некоторых альтернативных подходов, поэтому DQN может сходиться к Q^* не во всех случаях.

То есть даже при том, что иногда агенты DQN показывают очень высокую эффективность, они не являются универсальным решением для любых задач.

В завершение этой главы о глубоком обучении с подкреплением кратко перечислим типы агентов, помимо DQN. В числе основных категорий агентов глубокого обучения с подкреплением, как показано на рис. 13.11:

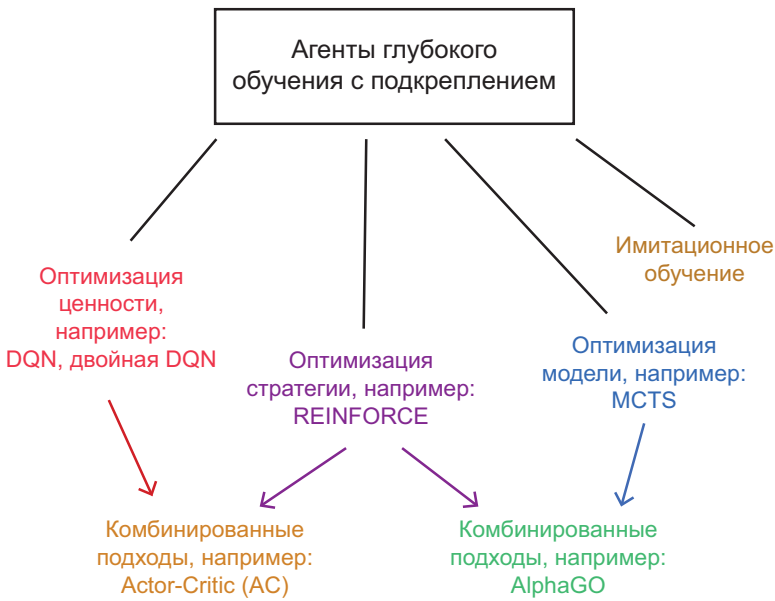


Рис. 13.11. Основные категории агентов глубокого обучения с подкреплением

- *Оптимизация ценности*. К этой категории относятся агенты DQN и их производные (например, *двойная DQN*, *дуэльная QN*), а также другие типы агентов, которые решают задачи обучения с подкреплением путем оптимизации функций ценности (включая функции Q-ценности).
- *Имитационное обучение*. Агенты из этой категории, например алгоритмы *копирования поведения* (behavioral cloning) и *условного имитационного обучения* (conditional imitation learning), обучаются имитировать поведение посредством демонстрации, Например, им показывают, как поставить тарелки на подставку для посуды или как налить воду в чашку. Хотя имитационное обучение является довольно увлекательным подходом, оно имеет относительно узкий диапазон применения, и мы не будем обсуждать его дальше.
- *Оптимизация модели*. Агенты из этой категории учатся предсказывать будущие состояния на основе (s, a) на заданном временном шаге. Примером таких алгоритмов может служить поиск по дереву Монте-Карло (Monte Carlo Tree Search, MCTS), упоминавшийся в главе 4 при обсуждении модели AlphaGo.
- *Оптимизация стратегии*. Агенты из этой категории изучают *непосредственно* стратегию, то есть функцию стратегии π, показанную на рис. 13.5. Более подробно с этой категорией мы познакомимся в следующем разделе.

ГРАДИЕНТЫ СТРАТЕГИЙ И АЛГОРИТМ REINFORCE

Как было показано на рис. 13.5, целью агента обучения с подкреплением является выявление некоторой функции стратегии π, которая отображает пространство состояний S в пространство действий A. Агенты DQN и любые другие агенты, основанные на оптимизации ценности, выясняют π косвенным путем, оценивая функции ценности, такие как оптимальная Q-ценность, Q^*. Агенты оптимизации стратегии, напротив, выясняют π *непосредственно*.

Особенно ярким представителем алгоритмов *градиентов стратегий* (Policy Gradient, PG), способных непосредственно выполнять градиентное *восхождение*[1], является хорошо известный алгоритм обучения с подкреплением REINFORCE[2]. Преимущество PG-алгоритмов, таких как REINFORCE, состоит в том, что они могут сходиться к довольно оптимальному решению[3], а значит,

---

[1] PG-алгоритмы *максимизируют* вознаграждение (вместо, скажем, минимизации стоимости), поэтому они выполняют градиентное *восхождение*, а не градиентный спуск. Подробнее об этом см. сноску в обсуждении уравнения 13.1 в этой главе.

[2] Williams, R. (1992). «Simple statistical gradient-following algorithms for connectionist reinforcement learning». *Machine Learning, 8*, 229–56.

[3] PG-агенты имеют свойство сходиться по крайней мере к оптимальному локальному решению, хотя было продемонстрировано, что некоторые конкретные методы PG способны находить оптимальное глобальное решение. Fazel, K., et al. (2018). «Global convergence of policy gradient methods for the linear quadratic regulator». *arXiv: 1801.05039*.

имеют более широкую область применения, чем алгоритмы оптимизации ценности, такие как DQN. Проблема, однако, в том, что PG-алгоритмы *менее последовательны*. То есть они имеют более высокую дисперсию по сравнению с подходами на основе оптимизации ценности, такими как DQN, и поэтому PG-алгоритмы, как правило, требуют большего количества обучающих образцов.

АЛГОРИТМ ACTOR-CRITIC

Как показано на рис. 13.11, алгоритм *actor-critic* (актор-критик) — это агент обучения с подкреплением, который объединяет подходы оптимизации ценности и оптимизации стратегии. В частности, как показано на рис. 13.12, он объединяет алгоритмы *Q*-обучения и PG. В общем случае такой алгоритм включает цикл, который чередует выполнение двух алгоритмов:

- *Актор*: PG-алгоритм, выбирающий действие.
- *Критик*: алгоритм *Q*-обучения, «критикующий» действие, выбранное актером, и сообщающий, как можно изменить это действие. Может использовать приемы увеличения эффективности, свойственные *Q*-обучению, такие как воспроизведение воспоминаний.

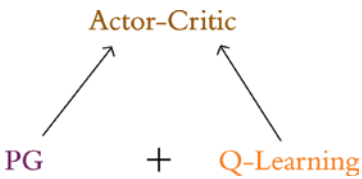


Рис. 13.12. Алгоритм actor-critic комбинирует алгоритм градиента стратегии в обучении с подкреплением (играет роль актера) и алгоритм *Q*-обучения (играет роль критика)

Алгоритм actor-critic имеет определенное сходство с генеративно-состязательными сетями из главы 12. Генеративно-состязательные сети включают сеть генератора и дискриминатора, первая из которых создает поддельные изображения, а вторая оценивает их. Алгоритм actor-critic заключает в себе актора и критика, первый из которых выполняет действия, а второй оценивает их.

Преимущество алгоритма actor-critic в том, что он может решать более широкий круг задач, чем DQN, и в то же время имеет меньшую дисперсию эффективности по сравнению с REINFORCE. Тем не менее из-за наличия внутри алгоритма PG actor-critic все же страдает некоторой неэффективностью.

Хотя реализация алгоритмов REINFORCE и actor-critic выходит за рамки этой книги, вы можете воспользоваться фреймворком SLM Lab и самостоятельно опробовать их, а также исследовать их реализацию.

ИТОГИ

В этой главе мы познакомились с теорией, лежащей в основе обучения с подкреплением, включая марковские процессы принятия решений. Воспользовавшись новыми знаниями, мы создали агента глубокого Q-обучения и обучили его игре в Cart-Pole. В заключение мы познакомились с другими алгоритмами глубокого обучения с подкреплением, помимо DQN, такими как REINFORCE и actor-critic, а также с SLM Lab — фреймворком глубокого обучения с подкреплением, включающим реализации некоторых алгоритмов и инструменты для оптимизации гиперпараметров агента.

Эта глава завершает третью часть книги, где мы представили вашему вниманию практические реализации моделей компьютерного зрения (глава 10), обработки естественного языка (глава 11), создания изображений (глава 12) и агентов принятия последовательностей решений. В следующей, заключительной части книги мы дадим подробное руководство по адаптации этих моделей для ваших собственных проектов и нужд.

КЛЮЧЕВЫЕ ПОНЯТИЯ

Вот основные ключевые понятия, с которыми мы познакомились к данному моменту. Новые понятия, описанные в этой главе, выделены серым:

- параметры:
  - вес w;
  - смещение b;
- активация a;
- искусственные нейроны:
  - sigmoid;
  - tanh;
  - ReLU;
  - Linear;
- входной слой;
- скрытый слой;
- выходной слой;
- типы слоев:
  - плотный (полносвязанный);
  - softmax;
  - сверточный;
  - развертывающий;

  - субдискретизации с объединением по максимальному значению;
  - повышения дискретизации;
  - преобразования в плоский массив;
  - создания векторных представлений;
  - рекуррентные;
  - (двунаправленные) LSTM;
  - объединения;
- функции стоимости (потерь):
  - квадратичная (среднеквадратичная ошибка);
  - перекрестная энтропия;
- прямое распространение;
- обратное распространение;
- нестабильность градиентов (в частности, затухание);
- метод Глоро инициализации весов;
- пакетная нормализация;
- прореживание;
- оптимизаторы:
  - стохастический градиентный спуск;
  - Adam;
- гиперпараметры оптимизаторов:
  - скорость обучения η;
  - размер пакета;
- word2vec;
- компоненты GAN:
  - сеть дискриминатора;
  - сеть генератора;
  - состязательная сеть;
- глубокое Q-обучение.

IV

ВЫ И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

14

ВПЕРЕД, К СОБСТВЕННЫМ ПРОЕКТАМ ГЛУБОКОГО ОБУЧЕНИЯ

Примите поздравления! Вы достигли заключительной главы книги! В первой части мы узнали, что такое глубокое обучение и как оно заняло лидирующие позиции. Во второй части углубились в теоретические основы глубокого обучения, а в третьей рассмотрели приемы применения теории для решения широкого круга задач, включающих зрение, язык, рисование и исследование изменяющегося окружения.

В этой главе мы представим вам ресурсы и советы, которые помогут перейти от примеров, показанных в третьей части, к созданию своих собственным проектов глубокого обучения, которые могут принести огромную пользу обществу. В ее конце расскажем, как ваша работа может способствовать дальнейшему развитию программного обеспечения для глубокого обучения в глобальном масштабе и, возможно, даже появлению универсального искусственного интеллекта.

ИДЕИ ДЛЯ ПРОЕКТОВ ГЛУБОКОГО ОБУЧЕНИЯ

В этом разделе рассматриваются идеи для ваших первых проектов глубокого обучения.

КОМПЬЮТЕРНОЕ ЗРЕНИЕ И ГЕНЕРАТИВНО-СОСТЯЗАТЕЛЬНЫЕ СЕТИ

Самым простым способом окунуться в мир практического применения глубокого обучения может стать загрузка набора данных Fashion-MNIST[1]. Библиотека Keras уже включает этот набор данных, состоящий из фотографий 10 классов одежды (табл. 14.1). Изображения в наборе Fashion-MNIST имеют те же раз-

[1] *Xiao H.* et al. (2017). «Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms». *arXiv: 1708.07747*.

меры, что и изображения цифр в наборе MNIST, с которыми вы познакомились во второй части книги: это черно-белые изображения 28 × 28 пикселов (одно из них показано для примера на рис. 14.1), в том числе 60 000 обучающих и 10 000 проверочных изображений. То есть заменив следующим кодом строку загрузки данных (например, в листинге 5.2) в любом блокноте Jupyter из этой книги, реализующем классификацию цифр MNIST, можно легко и просто перейти к задаче классификации данных из набора Fashion-MNIST:

```
from keras.datasets import fashion_mnist
(X_train, y_train), (X_valid, y_valid) = fashion_mnist.load_data()
```

Таблица 14.1. Категории в наборе данных Fashion-MNIST

Метка класса	Описание
0	футболки
1	брюки
2	свитеры
3	платья
4	пиджаки
5	сандалии
6	рубашки
7	тапки
8	сумки
9	ботинки

После этого вы можете начать экспериментировать с архитектурой моделью и настройками гиперпараметров для увеличения точности на проверочных данных. Изображения в наборе Fashion-MNIST гораздо сложнее классифицировать, чем рукописные цифры в наборе MNIST, поэтому, решая задачу их классификации, вы сможете в полной мере использовать знания, полученные в этой книге. В главе 10 мы добились более чем 99% точности на проверочных данных в задаче классификации данных из набора MNIST (см. рис. 10.9), но получить точность классификации Fashion-MNIST более 92% очень непросто, а результат, превышающий 94%, можно смело назвать весьма впечатляющим.

Вот еще несколько интересных ресурсов, где можно найти наборы данных для задач классификации изображений с использованием моделей глубокого обучения:

- *Kaggle*: эта платформа для проведения состязаний в области исследования данных имеет множество ценных наборов данных. Победа в состязаниях может принести реальные деньги! Например, в состязании «Cdiscount Image Classification Challenge» был учрежден денежный приз в размере 35 000 долларов США за классификацию изображений продуктов для

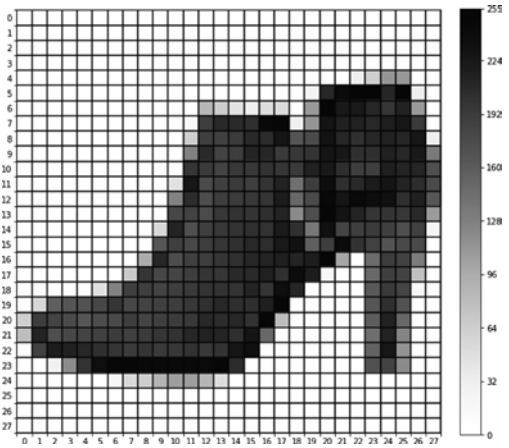


Рис. 14.1. По аналогии с изображениями рукописных цифр в наборе MNIST (рис. 5.3), набор Fashion-MNIST содержит изображения предметов одежды (одно из которых показано здесь). Это конкретное изображение относится к классу 9, то есть согласно табл. 14.1, это ботинок. Загляните в блокнот Jupyter fashion_mnist_pixel_by_pixel.ipynb, где приводится код, который мы использовали для создания этого рисунка

французского гиганта электронной коммерции[1]. Наборы данных, доступные в Kaggle, появляются и исчезают с началом и окончанием состязаний, но в любой конкретный момент времени вы найдете здесь большие наборы изображений, которые помогут получить опыт создания моделей, получить признание и, может быть, даже денежные призы.

- *Figure Eight*: эта компания, занимающаяся маркировкой данных через краудсорсинг (ранее известная как CrowdFlower), предоставляет десятки общедоступных и тщательно подобранных наборов данных для задач классификации изображений. Чтобы узнать, какие данные доступны, посетите сайт *figure-eight .com/data-for-everyone* и выполните поиск по слову *image*.
- Исследователь Люк де Оливейра (Luke de Oliveira) составил ясный и краткий список самых известных среди практиков глубокого обучения наборов данных. Загляните в раздел «Computer Vision» (компьютерное зрение) на bit.ly/LukeData.

Желающие создать и настроить свою генеративно-состязательную сеть могут начать с небольших наборов данных, например:

- Набор, включающий один или несколько классов изображений из набора данных Quick, Draw!, который мы использовали в главе 12[2].
- Набор Fashion-MNIST.
- Старый добрый набор изображений рукописных цифр MNIST.

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

Подобно набору данных Fashion-MNIST, который легко можно использовать для обучения моделей классификации изображений, созданных нами в этой книге,

[1] bit.ly/kaggleCD

[2] github.com/googlecreativelab/quickdraw-dataset

существуют наборы данных, подготовленные Сяном Чжаном (Xiang Zhang) и его коллегами из лаборатории Яна Лекуна (см. рис. 1.9), которые можно напрямую использовать для обучения моделей классификации естественного языка из главы 11, что делает их идеальным выбором для первого собственного проекта NLP.

Все восемь наборов данных на естественном языке, сформированных Чжаном и его коллегами, подробно описаны в их статье[1] и доступны для загрузки по адресу bit.ly/NLPdata. Каждый набор данных по крайней мере на порядок превосходит по объему набор данных IMDb с обучающей выборкой в 25 000 образцов, с которыми мы работали в главе 11, что позволяет экспериментировать с более сложными моделями глубокого обучения и конструировать гораздо более богатые векторные пространства слов. Шесть наборов данных из этой коллекции имеют более двух классов (поэтому в выходном слое должно иметься несколько нейронов с активацией softmax), а два других предназначены для задач бинарной классификации (что позволит вам сохранить один сигмоидный выход, как в наших примерах с данными IMDb):

- *Yelp Review Polarity*: 560 000 обучающих и 38 000 проверочных образцов, которые классифицируются как положительные (четырех- или пятизвездочные) или отрицательные (одно- или двухзвездочные) отзывы, размещенные на веб-сайте Yelp.
- *Amazon Review Polarity*: колоссальное число обучающих (3.6 миллиона) и проверочных (400 000) образцов отзывов, собранных гигантом электронной коммерции Amazon, которые делятся на два класса — положительные и отрицательные.

Данные для задач обработки естественного языка доступны также в Kaggle, Figure Eight (выполните поиск по слову *sentiment* или *text* на *figure-eight.com/data-for-everyone*) и на странице Люка де Оливейры (Luke de Oliveira, по адресу bit.ly/LukeData под заголовком «Natural Language»), которые можно использовать как основу для своих проектов глубокого обучения.

ГЛУБОКОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ

В качестве первого проекта глубокого обучения в этом направлении можно попробовать:

- *Изменить окружение*: например, обучить агента DQN из блокнота *cartpole_dqn.ipynb*[2], которого мы создали в главе 13, действовать в другом окружении из числа имеющихся в OpenAI Gym. Вот несколько относительно простых вариантов: Mountain Car (`MountainCar-v0`) и Frozen Lake (`FrozenLake-v0`).

---

[1] См. четвертый раздел («Large-scale Datasets and Results») в статье: *Zhang X.* et al. (2016). «Character-level convolutional networks for text classification». *arXiv: 1509.01626*.

[2] Для этого измените строковый аргумент, который передается в вызов `gym.make()` в листинге 13.1.

- *Создать нового агента*: если у вас есть доступ к компьютеру с ОС Unix (в том числе и macOS), вы можете установить SLM Lab (см. рис. 13.10), чтобы опробовать других агентов (например, Actor-Critic; см. рис. 13.12). Некоторые из них достаточно сложны, чтобы показать превосходные результаты в продвинутых окружениях, таких как игры для Atari[1] из библиотеки OpenAI Gym или трехмерные окружения из Unity.

Освоив работу с продвинутыми агентами, попробуйте обучить их в других окружениях, таких как DeepMind Lab (см. рис. 4.14), или обучить одного агента сразу в нескольких разных (SLM Lab поможет вам в этом).

ПРЕОБРАЗОВАНИЕ ИМЕЮЩЕГОСЯ ПРОЕКТА МАШИННОГО ОБУЧЕНИЯ

Все проекты, предложенные выше, связаны с использованием сторонних источников данных, но вы вполне можете собрать данные сами. Возможно, у вас даже уже есть какие-то данные, которые вы использовали для машинного обучения, например, в моделях линейной регрессии или в методе опорных векторов. Если это так, то можете попробовать использовать имеющиеся данные в некоторой модели глубокого обучения. Начните с полносвязанной сети, включающей три

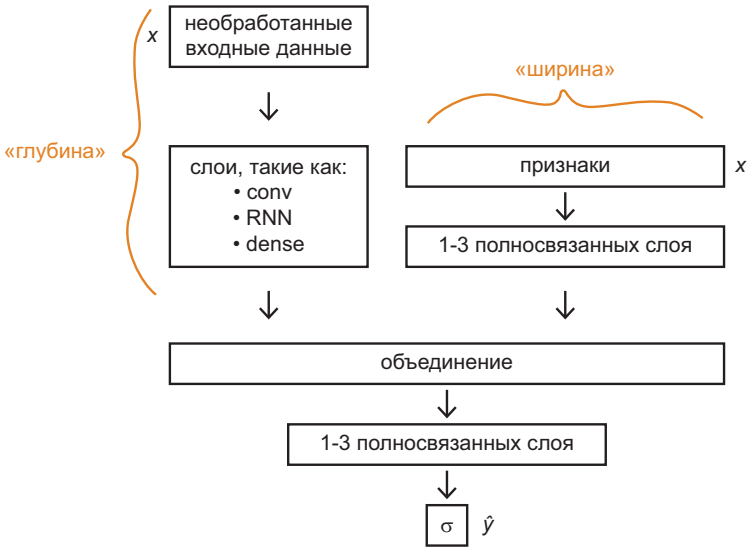


Рис. 14.2. Архитектура модели широкого и глубокого обучения объединяет результаты двух отдельных ветвей. Глубокая ветвь получает исходные (необработанные) данные и использует лишь несколько соответствующих слоев (например, сверточных, рекуррентных, полносвязанных) для автоматического извлечения признаков. Ее «глубина» определяется количеством имеющихся в ней слоев. Широкая ветвь получает признаки, извлеченные вручную (из исходных данных до моделирования с помощью экспертов). Ее «ширина» определяется количеством таких признаков

[1] gym.openai.com/envs/#atari

скрытых слоя, такой как в блокноте *deep_net_in_keras.ipynb* из главы 9. Если вам нужно предсказать непрерывную переменную, а не категориальную, то используйте в качестве шаблона наш блокнот *regression_in_keras.ipynb* (представлен в конце главы 9).

Вы можете попробовать передать в модель исходные данные в необработанном виде или, если у вас имеются уже извлеченные признаки, передать на вход эти признаки. Исследователи из Google[1] представили методику *широкого и глубокого обучения*, согласно которой модель одновременно обрабатывает найденные прежде и выявляет новые признаки в необработанных входных данных. На рис. 14.2 показана обобщенная схема этого подхода, включающего создание объединяющего слоя, который был представлен в конце главы 11 (см. листинг 11.41).

РЕСУРСЫ ДЛЯ БУДУЩИХ ПРОЕКТОВ

Для тех, кто пожелает выйти за рамки проектов начального уровня, предложенных выше, мы поддерживаем каталог полезных ресурсов по адресу jonkrohn.com/resources. Там вы найдете ссылки на:

- хорошо организованные общедоступные источники данных, иногда очень большие;
- рекомендуемые конфигурации аппаратной и облачной инфраструктуры для обучения крупных моделей глубокого обучения;
- подборки ключевых статей по глубокому обучению и реализации описываемых в них исследований;
- интерактивные демонстрационные примеры моделей глубокого обучения;
- примеры рекуррентных нейронных сетей для прогнозов по временным рядам, таким как финансовые данные[2].

СОЦИАЛЬНО ЗНАЧИМЫЕ ПРОЕКТЫ

Нам особенно хотелось бы привлечь ваше внимание к разделу «Problems Worth Solving» («Проблемы, заслуживающие внимания») на нашей странице с подборкой ресурсов. В этом разделе перечислены ресурсы, в которых кратко изложены наиболее острые глобальные проблемы, стоящие перед современным обществом, — проблемы, для решения которых вы могли бы применить методы глубокого обучения. Например, в одном из этих исследований[3] авторы из ис-

---

[1] bit.ly/wideNdeep

[2] Эта тема представляет большой интерес для многих студентов, изучающих глубокое обучение, но ее обсуждение выходит за рамки вопросов, рассматриваемых в этой книге.

[3] *Chui M.* (2018). «Notes from the AI frontier: Applying AI for social good». McKinsey Global Institute. bit.ly/aiForGood

следовательской организации McKinsey Global Institute представляют 10 социально значимых областей:

1. Равноправие и участие в общественной жизни.
2. Образование.
3. Борьба с болезнями и голодом.
4. Безопасность и правосудие.
5. Проверка и подтверждение достоверности информации.
6. Антикризисные меры.
7. Расширение экономических возможностей.
8. Государственный и социальный сектор.
9. Окружающая среда.
10. Инфраструктура.

Они подробно описывают потенциальную пригодность многих методов, представленных в этой книге, для решения проблем в каждой из перечисленных областей, в том числе:

- *глубокое обучение на структурированных данных* (полносвязанные сети, описанные в главах с 5-й по 9-ю): применимо ко всем 10 областям знаний;
- *классификация изображений*, включая *распознавание рукописного ввода* (глава 10): все области, кроме государственного и социального секторов;
- анализ естественного языка, включая определение *эмоциональной окраски* (глава 11): все области, кроме инфраструктуры;
- *создание контента* (глава 12): применимо к области равноправия и участия в общественной жизни, а также к области государственного и социального секторов;
- обучение с подкреплением (глава 13): применимо к области борьбы с болезнями и голодом.

ПРОЦЕСС МОДЕЛИРОВАНИЯ, ВКЛЮЧАЯ НАСТРОЙКУ ГИПЕРПАРАМЕТРОВ

При реализации любой из идей, представленных в этой главе, настройка гиперпараметров может оказаться важнейшим условием достижения успеха. В этом разделе мы опишем пошаговый процесс моделирования, который можно использовать как примерный шаблон при реализации своих проектов. Но имейте в виду, что из-за уникальных особенностей конкретного проекта вам может потребоваться отклониться от рекомендуемого процедурного пути. Например, маловероятно,

что вы будете проходить следующие этапы строго линейно: по достижении определенной точки в процессе реализации у вас может появиться догадка[1] о том, как можно усовершенствовать предыдущие этапы в зависимости от поведения модели[2], и в результате придется вернуться назад и повторить некоторые шаги несколько (иногда десять и более) раз! Вот наше пошаговое руководство:

1. *Инициализация параметров*: как рассказывается в главе 9 (см. рис. 9.3), параметры модели следует инициализировать разумными случайными значениями. Смещения мы рекомендуем инициализировать нулем, а для инициализации весов использовать метод Ксавье Глоро (Xavier Glorot). К счастью, при использовании библиотеки Keras такая разумная инициализация слоев, как правило, выполняется автоматически.

2. *Выбор функции стоимости*: при решении задач классификации обычно предпочтительнее использовать функцию стоимости перекрестной энтропии. В задачах регрессии лучше использовать среднеквадратичную ошибку. А желающие поэкспериментировать смогут найти другие варианты в `keras.io/losses`.

3. *Увеличение шансов на успех*: если первоначальная модель (например, основанная на любой из моделей из этой книги) показывает низкую эффективность на проверочных данных (например, точность <10% на данных из набора MNIST с 10 классами), можно попробовать следующую тактику:

   - *Упростить задачу*: например, при работе с цифрами MNIST можно уменьшить количество классифицируемых категорий с 10 до 2.
   - *Упростить архитектуру сети*: возможно, вы делаете что-то зря, не понимая этого. Или, может быть, модель слишком глубокая, из-за чего возникает сильный эффект затухания градиента. Модель с более простой архитектурой, например с меньшим числом слоев, может помочь выявить эти потенциальные проблемы.
   - *Уменьшить размер обучающего набора.* При использовании большого набора обучающих данных ожидание окончания одной эпохи может занять много времени. Уменьшив размер обучающей выборки, вы сможете совершать итерации и улучшать свою модель намного быстрее.

4. *Слои*: после обучения модели до некоторой степени можно начать экспериментировать со слоями. Например, можно попробовать:

---

[1] Поведение модели можно изучить, например, наблюдая за изменением потерь на обучающих и проверочных данных в процессе обучения модели. Это легко сделать с помощью TensorBoard (см. рис. 9.8).

[2] По мере накопления опыта в реализации проектов глубокого обучения и знакомства с высокоэффективными архитектурами других исследователей (например, по статьям в *GitHub*, *StackOverflow* и *arXiv*) вы будете понимать, как приспособить архитектуру и гиперпараметры вашей модели к данной задаче.

- *Изменить количество слоев*: следуя рекомендациям в главе 8 (рис. 8.8), можно попробовать добавлять или удалять отдельные слои или блоки слоев (например, блоки свертки-объединения на рис. 10.10).
- *Изменить типы слоев*: в зависимости от конкретной задачи и набора данных, одни типы слоев могут заметно превосходить другие. Вспомните, например, как влияло изменение типа слоя в нашем классификаторе эмоциональной окраски в главе 11 (см. табл. 11.6).
- *Изменить ширину слоя*: попробуйте изменить число нейронов в слое, используя значения, кратные степеням двойки, как было показано в главе 8 рядом с рис. 8.8.

5. *Предотвращение переобучения*: как рассказывалось в главе 9, развивайте способность модели обобщать данные, которые она прежде не видела, используя прореживание, обогащение данных (если это возможно, как, например, в случае с изображениями) и/или пакетную нормализацию. Если удастся получить дополнительные обучающие данные для вашей модели, это, вероятно, тоже пойдет ей на пользу. Наконец, как не раз было показано в главе 11, если модель страдает проблемой переобучения, можно попробовать повторно загрузить веса, полученные в одной из предыдущих эпох — той, в которой потери на проверочных данных были самыми низкими (рис. 14.3).

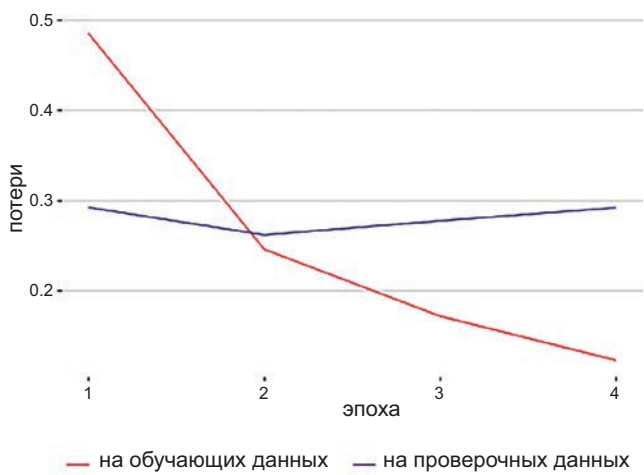


Рис. 14.3. График изменения потерь на обучающих (красный) и проверочных (синий) данных на протяжении нескольких эпох обучения. Эти конкретные результаты взяты из нашего блокнота multi_convnet_sentiment_classifier.ipynb (см. последний раздел главы 11), но тот же эффект переобучения типичен для моделей глубокого обучения. После эпохи 2 потери на обучающих данных продолжают приближаться к нулю, а потери на проверочных данных ползут вверх. Эпоха 2 имеет наименьшие потери на проверочных данных, поэтому ее параметры следует загрузить повторно для дальнейшего тестирования модели и (возможно!) даже для использования в промышленной системе

6. *Скорость обучения*: попробуйте уменьшать и увеличивать скорость обучения, как было описано в главе 9. Но имейте в виду, что «необычным» оптимизаторам, таким как Adam и RMSProp, часто удается автоматически регулировать скорость обучения[1].
7. *Размер пакета*: этот гиперпараметр является, пожалуй, одним из наименее важных, поэтому его настройку можно отложить на потом. Рекомендации вы найдете в главе 8 (рядом с рис. 8.7).

АВТОМАТИЗАЦИЯ ПОИСКА ГИПЕРПАРАМЕТРОВ

Экспериментировать с гиперпараметрами любой данной модели глубокого обучения можно до бесконечности, поэтому неудивительно, что разработчики (которые очень ленивы!) придумали методы автоматизации поиска гиперпараметров. В главе 13 мы рассмотрели использование SLM Lab для подбора гиперпараметров в моделях глубокого обучения с подкреплением; для оптимизации гиперпараметров мы советуем использовать Spearmint[2]. Но имейте в виду, что независимо от выбранного подхода к подбору гиперпараметров, Джеймс Бергстра и Йошуа Бенжио[3] из Монреальского университета представили доказательства, что метод случайного выбора значений для гиперпараметров с большей вероятностью позволяет найти оптимальные гиперпараметры, чем поиск по жестко структурированной сетке; см. рис. 14.4[4].

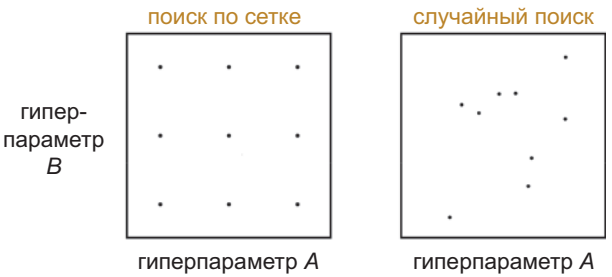


Рис. 14.4. Поиск по сетке (слева) с меньшей вероятностью позволяет найти оптимальные гиперпараметры для данной модели, чем случайный поиск (справа)

---

[1] Обратите внимание, что из этого правила есть исключения. Например, мы обнаружили в главах 12 (в отношении GAN) и 13 (в отношении агентов обучения с подкреплением), что настройка скорости обучения эффективна даже с такими оптимизаторами, как Adam и RMSProp.

[2] *Snoek J.* et al. (2012). «Practical Bayesian optimization of machine learning algorithms». Advances in Neural Information Processing Systems, 25. Код доступен по адресу github.com/JasperSnoek/spearmint.

[3] Портрет Бенжио вы найдете на рис. 1.10.

[4] *Bergstra J. & Bengio Y.* (2012). «Random search for hyper-parameter optimization». Journal of Machine Learning Research, 13, 281–305.

БИБЛИОТЕКИ ГЛУБОКОГО ОБУЧЕНИЯ

На протяжении всей книги для построения и запуска моделей глубокого обучения мы использовали библиотеку Keras. Однако существует масса других библиотек глубокого обучения, и каждый год продолжают появляться новые. В этом разделе мы рассмотрим несколько основных альтернатив.

KERAS И TENSORFLOW

TensorFlow является, пожалуй, самой известной библиотекой глубокого обучения. Ее название происходит от идеи *тензоров* (*tenzor*, массивов информации, например, входных данных x модели или активаций a), *протекающих* (*flow*) через последовательность операций (например, определяющих математику искусственных нейронов, таких как наше «самое важное уравнение» на рис. 6.7). Первоначально библиотека TensorFlow разрабатывалась в Google для внутреннего использования, но в 2015 году технический гигант открыл исходный код проекта. На рис. 14.5 показано, как менялся интерес к пяти наиболее популярным библиотекам глубокого обучения, если судить по относительной частоте поисковых запросов в Google. Библиотека Keras заняла второе место, а TensorFlow является безусловным лидером. Учитывая это, у вас может появиться желание узнать, как использовать TensorFlow. Если это так, у нас есть хорошие новости для вас: вы уже знаете, как это сделать.

Библиотека Keras — это не только высокоуровневый API-интерфейс, который мы использовали в этой книге для вызова функций из TensorFlow, но также — с момента выхода TensorFlow 2.0 в 2019 году — рекомендуемый в документации

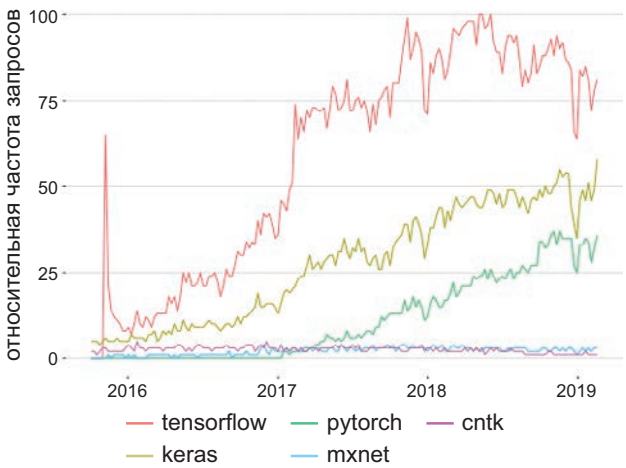


Рис. 14.5. Изменение интереса к пяти наиболее популярным библиотекам глубокого обучения (на основе относительной частоты поисковых запросов в Google с октября 2015-го по февраль 2019-го)

к самой TensorFlow способ построения моделей с применением слоев Keras. Раньше для создания моделей с использованием TensorFlow необходимо было освоить довольно сложный трехэтапный процесс:

1. Сконструировать подробный вычислительный граф.
2. Инициализировать его в рамках сеанса.
3. Передать данные в сеанс и извлечь необходимую информацию (например, сводные метрики, параметры модели) из сеанса.

Этот довольно сложный и запутанный процесс позволяет библиотеке TensorFlow оптимизировать обучение модели и ее использование на этапе эксплуатации с привлечением всех доступных устройств (процессоров и графических сопроцессоров, возможно, распределенных по нескольким серверам). Со временем разработчики таких библиотек, как PyTorch, создали остроумные механизмы, помогающие взять лучшее из обоих миров:

1. Концептуально простая и быстрая процедура создания многослойных моделей глубокого обучения *и одновременно*...
2. Высокооптимизированное выполнение моделей на всех доступных устройствах.

Команда TensorFlow отреагировала на это более тесной интеграцией с Keras и созданием *оперативного режима* (eager mode), обеспечивающего немедленное выполнение (вместо прежде использовавшегося трехэтапного процесса) без ущерба для эффективности. До появления TensorFlow 2.0 этот режим требовалось активировать вручную[1], но начиная с версии 2.0 этот режим используется по умолчанию.

Теперь не представляет большого труда преобразовать любой код, который мы рассмотрели в этой книге и использующий библиотеку Keras, так, чтобы он использовал только библиотеку TensorFlow. Например, загляните в наш блокнот *deep_net_in_tensorflow.ipynb*, который идентичен блокноту *deep_net_in_keras.ipynb* (из главы 9) и отличается только перечнем зависимостей (сравните листинг 14.1 с листингами 5.1 и 9.4).

Листинг 14.1. Зависимости, необходимые для построения глубокой сети со слоями Keras и с использованием TensorFlow без подключения самой библиотеки Keras

```
import tensorflow as tf
from tensorflow.python.keras.datasets import mnist
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Dropout
from tensorflow.python.keras.layers import BatchNormalization
from tensorflow.python.keras.optimizers import SGD
from tensorflow.python.keras.utils import to_categorical
```

[1] Единственной строкой кода: `tf.enable_eager_execution()`.

Теперь вы можете начать исследовать дополнительные возможности TensorFlow.

В частности, TensorFlow со слоями Keras можно использовать вместо высокоуровневого Keras API:

- для настройки модели в соответствии с вашими пожеланиями, в том числе путем *создания подкласса* `tf.keras.Model` для организации прямого распространения данных в вашей модели любым способом, каким только пожелаете[1];
- для создания высокопроизводительных конвейеров ввода данных с использованием `tf.data`;
- для развертывания моделей в:
  - высокопроизводительных системах на серверах с TensorFlow Serving;
  - мобильных и встроенных устройствах с TensorFlow Lite;
  - веб-браузерах с TensorFlow.js.

PYTORCH

Библиотека PyTorch — близкая родственница системы машинного обучения под названием Torch, основанной на языке программирования Lua. Фактически PyTorch — это расширение для системы Torch, разработанное с целью дать простую и понятную возможность ее использования из более широко распространенного языка Python. В основном PyTorch разрабатывалась в подразделении Facebook AI Research под руководством Яна Лекуна (см. рис. 1.9). Даже при том, что PyTorch не так популярна, как TensorFlow или Keras, она стала существенно влиятельнее за короткий период времени (см. рис. 14.5), и это вполне объяснимо, как мы покажем ниже.

Многие высокоуровневые библиотеки глубокого обучения (включая Keras) являются простыми обертками для низкоуровневого кода (на Python и на других языках, таких как C); однако PyTorch — это не просто обертка на Python для Torch. Она была написана с нуля, специально для людей, знакомых с Python, и при этом сохранила вычислительную эффективность оригинальной библиотеки Torch.

В своей основе PyTorch выполняет матричные операции так же, как NumPy. В действительности тензоры PyTorch совместимы с большинством операций, реализованных в NumPy, а кроме того, существуют методы преобразования между массивами NumPy и тензорами PyTorch. Благодаря такой глубокой интеграции с NumPy слои могут быть написаны непосредственно на Python, если потребуется дополнительная гибкость. Однако в отличие от NumPy, библиотека PyTorch имеет специальные механизмы для вычислений на графических процессорах, то есть может использовать способность этих процессоров выполнять

[1] См. tensorflow.org/guide/keras#model_subclassing.

параллельно массивные матричные операции. Кроме того, в PyTorch имеются встроенные оптимизированные библиотеки, помогающие производить вычисления быстро, независимо от устройства, а настраиваемые механизмы распределения памяти позволяют максимально эффективно использовать память.

Желающие узнать больше найдут в приложении C перечень многих функций из библиотеки PyTorch. Мы сравним их с функциями из TensorFlow и представим практический пример модели глубокого обучения. Как вы увидите, синтаксис PyTorch аналогичен синтаксису Keras, и вы быстро его освоите, если захотите.

MXNET, CNTK, CAFFE И ДРУГИЕ

Кроме Keras, TensorFlow и PyTorch существует множество других библиотек глубокого обучения. Вот некоторые из них:

- MXNet, разработанная в Amazon.
- CNTK, или Microsoft Cognitive Toolkit.
- Caffe, разработанная в Университете Беркли исключительно для систем компьютерного зрения и приложений на основе сверточных нейронных сетей. Caffe2 — упрощенная преемница, разработанная в Facebook AI Research, но в 2018 году эта библиотека была включена в проект PyTorch.
- Theano, проект Монреальского университета, некогда соперничавший с TensorFlow за звание ведущей библиотеки глубокого обучения. В настоящее время не развивается, в основном потому, что многие из разработчиков перешли в проект Google TensorFlow.

Все эти библиотеки — и другие, пользующиеся популярностью, — распространяются с открытым исходным кодом. Кроме того, подавляющее большинство этих библиотек следуют за дизайном Keras, основное внимание уделяя слоям разного типа, и потому имеют схожий синтаксис. У вас не должно возникнуть проблем с их использованием, если появится такая необходимость.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ 2.0

Модели, создание которых облегчают все доступные библиотеки глубокого обучения, радикально меняют мир программного обеспечения. В своей широко известной статье выдающийся исследователь данных Андрей Карпатый (рис. 14.6) утверждает, что появление глубокого обучения ознаменовало начало эпохи «программного обеспечения 2.0». К программному обеспечению 1.0 Карпатый относит классические языки программирования, такие как Python, Java, JavaScript, C ++ и другие. В программном обеспечении 1.0 мы должны описать в компьютерной программе четкие инструкции, чтобы компьютер смог вычислить желаемые результаты.

Рис. 14.6. Андрей Карпатый — директор подразделения искусственного интеллекта в калифорнийской автомобильной и энергетической компании Tesla. Мы уже упоминали его выше — в сноске в главе 10. Карпатый имеет опыт работы во многих организациях, упоминавшихся в этой книге, в том числе в OpenAI (см. рис. 4.13 и главу 13), Стэнфордском университете (где он защитил кандидатскую диссертацию под руководством Фей-Фей Ли (см. рис. 1.14), DeepMind (например, рисунки с 4.4 по 4.10), Google (многократно упоминавшейся в этой книге, в том числе в роли разработчика TensorFlow) и Университете Торонто (например, рис. 1.16 и рис. 3.2)

Программное обеспечение 2.0, напротив, состоит из моделей глубокого обучения, которые аппроксимируют функции, подобные тем, что мы аппроксимировали в этой книге для классификации рукописных цифр, прогнозирования цен на жилье, анализа эмоциональной окраски отзывов к фильмам, генерирования рисунков яблок и поиска оптимальной стратегии Q^* в игре Cart-Pole. Миллионы и миллиарды параметров в получающихся моделях глубокого обучения сегодня во все большей степени демонстрируют себя как более адаптируемые, полезные и мощные, чем жестко запрограммированные функции в мире программного обеспечения 1.0. Программное обеспечение 2.0 не заменяет программное обеспечение 1.0: первое основывается на втором — программное обеспечение 1.0 предоставляет всю цифровую инфраструктуру, в которой существует программное обеспечение 2.0.

Вот некоторые из основных преимуществ программного обеспечения 2.0, перечисленных в работе Карпатого:

1. *Вычислительная однородность*: модели глубокого обучения состоят из однородных единиц, таких как нейроны ReLU, что позволяет оптимизировать и масштабировать матричные вычисления с их участием.
2. *Постоянное время выполнения*: после переноса в промышленное окружение модель глубокого обучения будет выполнять одинаковый объем вычислений, независимо от входных данных, вводимых в нее. Подходы, на которых базируется программное обеспечение 1.0, могут включать в себя бесчисленные операторы if-else и требовать разного объема вычислений в зависимости от особенностей входных данных.
3. *Постоянное потребление памяти*: по аналогии с временем выполнения, модели глубокого обучения в промышленном окружении потребляют один и тот же объем памяти, независимо от входных данных.
4. *Простота*: прочитав эту книгу, вы приобрели навыки создания высокопроизводительных алгоритмов в самых разных областях. До появления глубокого обучения требовалось иметь значительно больше знаний в конкретных областях, чтобы реализовать то же самое в каждой отдельной области.
5. *Превосходство*: как мы расскажем в следующем абзаце, модели глубокого обучения могут значительно превосходить другие подходы.

Рассмотрим в свете этих аспектов приложения, которые были описаны в третьей части этой книги:

- *Компьютерное зрение* (например, распознавание цифр MNIST в главе 10): в традиционном машинном обучении для этого требовалось заранее определить визуальные признаки, для чего, как правило, необходим многолетний опыт работы в этой области. Модели глубокого обучения действуют намного лучше (см. рис. 1.15): они автоматически выявляют признаки и не требуют каких-то особых специальных знаний для их реализации.
- *Обработка естественного языка* (например, анализ эмоциональной окраски в главе 11): в традиционном машинном обучении для создания эффективного алгоритма необходим многолетний опыт в лингвистике, в том числе и понимание уникального синтаксиса и семантики любого данного языка. Модели глубокого обучения, как правило, работают лучше (как показано на рис. 2.3). Они автоматически выявляют соответствующие признаки и не требуют, чтобы разработчик обладал большим лингвистическим опытом.
- *Имитация художественных и визуальных образов* (например, рисунков в главе 12): генеративно-состязательные сети, основанные на моделях глубокого обучения, создают гораздо более убедительные и реалистичные изображения, чем любые ранее существовавшие подходы[1].
- *Игра в игры* (например, сети глубокого *Q*-обучения в главе 13): единственный алгоритм, AlphaZero, способен превзойти любое программное обеспечение 1.0 или традиционное решение машинного обучения в игре Го, шахматах и Сёги (как показано на рис. 4.10). Примечательно, что он делает это более эффективно и не требует обучающих данных.

НА ПУТИ К УНИВЕРСАЛЬНОМУ ИСКУССТВЕННОМУ ИНТЕЛЛЕКТУ

Как рассказывалось в главе 1 о развитии зрения у трилобитов (см. рис. 1.1), прошло много миллионов лет, прежде чем животные приобрели сложную систему цветного зрения, которую мы можем использовать себе во благо. Напротив, чтобы первые системы компьютерного зрения (см. рис. 1.8) развились в решения, если не превосходящие, то, по крайней мере, не уступающие людям в задачах распознавания образов (см. рис. 1.15)[2], понадобилось всего несколько десятилетий. Даже при том что классификация изображений является классическим

[1] Посетите страницу distill.pub/2017/aia, где вы сможете ознакомиться с интерактивной статьей Шана Картера (Shan Carter) и Майкла Нильсена (Michael Nielsen), в которой рассказывается, как можно использовать генеративно-состязательные сети для расширения человеческого интеллекта.

[2] Кстати, эталоном человеческой точности на рис. 1.15 является сам Андрей Карпатый (рис. 14.6).

примером узкоспециализированного искусственного интеллекта (Artificial Narrow Intelligence, ANI), такое быстрое развитие позволяет надеяться, что *универсальный* искусственный интеллект (Artificial General Intelligence, AGI) и, возможно, даже искусственный суперинтеллект (Artificial Super Intelligence, ASI) появятся еще при нашей жизни[1]. Например, результаты опросов Мюллера и Бострома, упоминавшиеся в главе 1, дают в качестве медианных оценок появления универсального и суперинтеллекта 2040 и 2060 годы соответственно.

Быстрому прогрессу в ANI способствовали четыре основных фактора, и они же могут помочь в достижении AGI или ASI:

1. *Данные*: в последние годы объем цифровых данных удваивается примерно каждые 18 месяцев, и пока не наблюдается никаких признаков замедления этого экспоненциального роста (вспомните, например, о неумолимо увеличивающемся потоке данных, извлекаемом отдельным беспилотным автомобилем, как рассказывалось в главе 13). Большая часть данных имеет низкое качество, но, как и в случае открытых источников данных, упоминавшихся выше в этой главе, их наборы становятся больше, дешевле в хранении и часто лучше организованы (примером может служить набор ImageNet, упоминавшийся в главах 1 и 10).

2. *Вычислительная мощность*: в ближайшие годы скорость увеличения производительности отдельных процессоров может замедлиться, тем не менее массовое распараллеливание матричных операций в графических процессорах и между множеством серверов — каждый с несколькими процессорами и, возможно, с несколькими графическими процессорами — будет продолжать увеличивать доступную вычислительную мощность.

3. *Алгоритмы*: быстро растущая армия ученых и инженеров, профессионально занимающихся обработкой данных, — глобальных и разбросанных по академическим и коммерческим организациям, — совершенствует методы анализа наборов данных для выявления типичных шаблонов. Время от времени случаются такие прорывы, как появление AlexNet (см. рис. 1.15). Большинство этих прорывов, многие из которых мы рассмотрели в этой книге, стали возможны благодаря глубокому обучению.

4. *Инфраструктура*: инфраструктура программного обеспечения 1.0, например операционные системы с открытым исходным кодом и языки программирования, в сочетании с библиотеками и методами программного обеспечения 2.0 (которые распространяются по всему миру через arXiv и GitHub) и низкой стоимостью услуг облачных вычислений (например, Amazon Web Services, Microsoft Azure, Google Cloud Platform) обеспечивает обширное поле для экспериментов с более крупными наборами данных.

Когнитивные задачи, которые люди обычно считают трудными (например, игра в шахматы, решение задач матричной алгебры, оптимизация финансо-

[1] Вернитесь к концу главы 1, чтобы узнать больше о ANI, AGI и ASI.

вого портфеля), — это, как правило, те задачи, которые *Homo sapiens* решал на протяжении не более тысячи лет. И это те самые задачи, с которыми легко справляются современные компьютеры. Познавательные задачи, которые люди считают простыми (например, чтение социальных сигналов, безопасное перемещение ребенка вверх по лестнице), напротив, решались людьми на протяжении миллионов лет и сегодня остаются недосягаемыми для компьютеров. Поэтому, несмотря на все волнующие достижения в машинном обучении, мы еще очень далеки от появления универсального искусственного интеллекта (AGI), продолжающего оставаться лишь теоретической возможностью. Вот некоторые примеры существенных препятствий, мешающих появлению AGI[1]:

- Глубокое обучение требует анализа *многих, многих образцов*. Эти большие наборы данных не всегда доступны, тогда как биологическим системам обучения, в том числе у мышей и детей, часто достаточно одного примера.
- Модели глубокого обучения обычно представляют собой *черный ящик*. И хотя существуют такие методы исследования, как DeepViz[2], созданный Джейсоном Йосински и его коллегами, они являются скорее исключением из правил.
- Модели глубокого обучения не используют знания об окружающем мире; например, они не учитывают множество факторов при принятии решений.
- В моделях глубокого обучения предсказанная корреляция между некоторыми входом x и выходом y не используется для оценки причинно-следственной связи. Способность переходить от предсказания корреляции между переменными к выявлению причинно-следственных связей между ними, по-видимому, имеет решающее значение для развития универсального интеллекта.
- Модели глубокого обучения часто подвержены странным и непонятным ошибкам[3], и их можно намеренно обмануть, изменив всего один пиксел во входном изображении[4].

Возможно, задача преодоления некоторых из этих барьеров заинтересует вас, и вы сможете посвятить часть своей карьеры разработке решений! Мы не можем точно предсказать, что ждет нас в будущем, но, учитывая взрывное накопление данных, рост вычислительных мощностей, развитие алгоритмов и инфраструк-

---

[1] Подробнее об ограничениях глубокого обучения читайте в статье: Marcus, G. (2018). «Deep learning: A critical appraisal». *arXiv: 1801.00631*.

[2] bit.ly/DeepViz

[3] Печально известный пример можно найти на странице bit.ly/googleGaffe.

[4] Умышленное введение в заблуждение алгоритма машинного обучения называется *состязательной атакой* и осуществляется путем ввода *состязательного примера*. Есть много работ, посвященных этому вопросу; одна из них, описывающая однопиксельные состязательные атаки на сверточную сеть: *Su J.* et al. (2017). «One pixel attack for fooling deep neural networks». *arXiv: 1710.08864*.

туры, мы совершенно уверены, что у вас не должно возникнуть особых проблем в определении направлений для применения глубокого обучения.

ИТОГИ

Эта глава завершила книгу, предложив вашему вниманию некоторые идеи, ресурсы для дальнейшего изучения, общие рекомендации по обучению моделей, обзор моделей глубокого обучения, доступных за рамками Keras, и способов быстрого изменения программного обеспечения под влиянием искусственных нейронных сетей, которые будут обладать большим потенциалом в ближайшие годы!

Удачи вам в будущем и оставайтесь с нами на связи:

- Аккаунт в Твиттере, который мы используем для публикации новых идей и достижений по мере их появления, включая новые видеоруководства, которые мы предполагаем опубликовать в дополнение к сведениям, описанным в этой книге: twitter.com/JonKrohnLearns.
- Для публикации длинных статей мы используем Medium: medium.com/@jonkrohn.
- Мы создали форум в Google Group, где читатели этой книги смогут задавать вопросы и отвечать друг другу (возможно, даже нам!). Вы найдете его по адресу bit.ly/DLIforum.
- И наконец, добавляйте нас в свои связи в LinkedIn (например, linkedin.com/in/jonkrohn), но *не забудьте упомянуть, что вы наш читатель*, потому что мы не принимаем запросы от посторонних!

Рис. 14.7. Трилобит машет вам рукой на прощание

Мы надеемся, что вам понравилось это иллюстрированное введение в глубокое обучение. Мы глубоко благодарны за время и силы, которые вы потратили на это путешествие вместе с нами. В добрый путь, дорогой друг, говорит вам трилобит на рис. 14.7!

V

ПРИЛОЖЕНИЯ

А

ФОРМАЛЬНАЯ НОТАЦИЯ НЕЙРОННЫХ СЕТЕЙ

Чтобы упростить обсуждение искусственных нейронов, мы использовали в этой книге сокращенную нотацию для их представления в сети. В этом приложении мы приводим более широко используемые формальные обозначения, которые кому-то из вас могут показаться интересными, потому что они:

- более точно описывают нейроны;
- близко следуют методике обратного распространения, описанной в приложении Б.

На рис. 7.1 изображена нейронная сеть, которая в общей сложности имеет четыре слоя. Первый — это входной слой, который можно рассматривать как набор начальных блоков, представляющий каждый образец входных данных. Например, в моделях MNIST имеется 784 таких начальных блока, представляющих пикселы в изображении рукописной цифры из набора MNIST размером 28×28 пикселов. Внутри входного слоя не выполняется никаких вычислений; он просто предоставляет пространство ввода исходных значений, чтобы сеть знала, сколько значений необходимо подготовить для вычислений на следующем уровне[1].

Следующие два слоя в сети на рис. 7.1 — это скрытые слои, в которых протекают основные вычисления. Как отмечается чуть ниже, каждый нейрон в скрытом слое математически преобразует и комбинирует входные значения x и выводит некоторое значение активации a. Поскольку нам нужно как-то обозначить каждый нейрон в каждом слое, будем использовать верхний индекс — для определения слоя, начиная с первого скрытого слоя, а нижний индекс для определения нейрона в этом слое. Соответственно, на рис. 7.1 в первом скрытом слое

[1] По этой причине нам обычно не требуется обозначать входные нейроны; у них нет ни весов, ни смещений.

мы имеем нейроны a_1^1, a_2^1 и a_3^1. Так можно сослаться на любой нейрон в любом слое. Например, a_2^2 представляет второй нейрон во втором скрытом слое.

Поскольку на рис. 7.1 изображена полносвязанная сеть, нейрон a_1^1 получает входные данные от всех нейронов из предыдущего слоя, то есть от входов x_1 и x_2 сети. Каждый нейрон имеет свое смещение, b. Мы будем обозначать эти смещения точно так же, как активации a. Например, b_2^1 — это смещение второго нейрона в первом скрытом слое.

Зеленые стрелки на рис. 7.1 представляют математические преобразования, выполняемые в ходе прямого распространения, и каждая из них имеет свой вес. Для ссылки на эти веса мы используем обозначение $w_{(1,2)}^1$ — это вес в *первом* скрытом слое (верхний индекс), связывающий нейрон a_1^1 со входом x_2 во входном слое (нижний индекс). Необходимость использовать такой двузначный нижний индекс обусловлена тем, что сеть является полносвязанной: каждый нейрон в каждом слое связан с каждым нейроном в предшествующем слое, и каждая связь имеет свой вес. Давайте обобщим запись весов:

- Верхний индекс — это номер скрытого слоя, в котором находится нейрон, получающий вход.
- Первый нижний индекс — это номер нейрона в скрытом слое, получающего вход.
- Второй нижний индекс — это номер нейрона в предыдущем скрытом слое, служащий источником данных.

Вот еще один пример: веса для нейрона a_2^2 будут обозначаться как $w_{(2,i)}^1$, где i — номер нейрона в предыдущем слое.

Наконец, крайний правый слой в сети на рис. 7.1 — это выходной слой. Как и в скрытых слоях, нейроны в нем имеют веса и смещения и обозначаются точно так же.

Б

ОБРАТНОЕ РАСПРОСТРАНЕНИЕ

В этом приложении мы используем формальную нотацию нейронных сетей, описанную в приложении А, и погрузимся в вычисления частной производной методом обратного распространения, описанным в главе 8.

Начнем с определения дополнительных обозначений, которые помогут нам в этом. Обратное распространение происходит в обратном направлении, поэтому обозначения основаны на отсчете от последнего слоя (обозначается как L), а более ранние слои обозначаются индексами, отсчитываемыми от него ($L-1$, $L-2$, ... $L-n$). Веса, смещения и выходы функций подписываются соответствующими нижними индексами. Согласно уравнениям 7.1 и 7.2, активация слоя a^L вычисляется умножением активации из предыдущего слоя (a^{L-1}) на весовые коэффициенты w^L и смещения b^L, чтобы получить значение z^L, которое затем передается в функцию активации (здесь обозначается просто как σ). Кроме того, в конце мы реализуем простую функцию стоимости с помощью евклидова расстояния. Соответственно, для последнего слоя имеем:

$$z^L = w^L \cdot a^{L-1} + b^L. \tag{Б.1}$$

$$a^L = \sigma\left(z^L\right). \tag{Б.2}$$

$$C_0 = \left(a^L - y\right)^2. \tag{Б.3}$$

В каждой итерации нужно вычислить градиент полной ошибки в предыдущем слое ($\partial C / \partial a^L$); именно так общая ошибка системы распространяется в обратном направлении. Мы обозначаем это значение как δ_L. Поскольку обратное распространение происходит от конца к началу, оно начинается с выходного слоя. Этот слой является особым случаем, потому что ошибка *возникает* здесь в форме функции стоимости, и над ней нет других слоев. Поэтому δ_L задается следующей формулой:

$$\delta_L = \frac{\partial C}{\partial a^L} = 2\left(a^L - y\right). \tag{Б.4}$$

Повторим еще раз, что это — особый случай вычисления начального значения δ; в остальных слоях вычисления выполняются иначе (подробнее об этом чуть ниже). Чтобы обновить веса в слое L, нужно найти градиент стоимости относительно весов $\partial C / \partial a^L = \delta_L$. Согласно цепному правилу, он равен произведению градиента стоимости для предыдущего слоя относительно его выхода, градиента функции активации относительно z и градиента z относительно весов w^L:

$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}. \tag{Б.5}$$

Поскольку $\partial C / \partial a^L = \delta_L$ (уравнение Б.4), это уравнение можно упростить до:

$$\frac{\partial C}{\partial w^L} = \delta_L \cdot a^{L-1}\left(1 - a^{L-1}\right) \cdot a^{L-1}. \tag{Б.6}$$

Это значение, по сути, является относительной величиной, посредством которой веса в слое L влияют на общую стоимость, и мы используем ее для корректировки весов в этом слое. Однако наша работа не завершена; теперь нужно продолжить обратное распространение через остальные слои. Для слоя $L - 1$:

$$\delta_{L-1} = \frac{\partial C}{\partial a^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}}. \tag{Б.7}$$

И снова $\partial C / \partial a^L = \delta_L$ (уравнение Б.4). Таким способом общая ошибка передается вниз, то есть *распространяется в обратном направлении*. Остальные члены имеют производные, поэтому уравнение можно упростить до:

$$\delta_{L-1} = \frac{\partial C}{\partial a^{L-1}} = \delta_L \cdot a^L\left(1 - a^L\right) \cdot w^{L-1}. \tag{Б.8}$$

Теперь нужно найти градиент стоимости относительно весов в слое $L - 1$, как и прежде:

$$\frac{\partial C}{\partial w^{L-1}} = \frac{\partial C}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}}. \tag{Б.9}$$

И снова, подставив δ_{L-1} вместо $\partial C / \partial a^{L-1}$ (уравнение Б.8) и взяв производные от других членов, получим:

$$\frac{\partial C}{\partial w^{L-1}} = \delta_{L-1} \cdot a^{L-1}\left(1 - a^{L-1}\right) \cdot a^{L-2}. \tag{Б.10}$$

Этот процесс повторяется слой за слоем, пока не будет достигнут первый.

Напомним, что сначала мы находим значение δ_L (уравнение Б.4) — ошибку функции стоимости (уравнение Б.3) — и используем его в уравнении про-

изводной функции стоимости относительно весов в слое L (уравнение Б.6). В следующем слое мы находим δ_{L-1} (уравнение Б.8) — градиент стоимости относительно выходов слоя $L - 1$. Это значение точно так же используется в уравнении вычисления градиента функции стоимости относительно весов в слое $L - 1$ (уравнение Б.10). И так далее; обратное распространение продолжается до достижения входа модели.

До сих пор в этом приложении мы имели дело только с сетями, имеющими один вход, один скрытый нейрон и один выход. Но на практике модели глубокого обучения никогда не бывают такими простыми. К счастью, уравнения, показанные выше, легко распространить на случай с несколькими нейронами в слое и несколькими входами и выходами.

Рассмотрим ситуацию, когда на выходе имеется несколько классов, как, например, в задаче классификации цифр MNIST. В этом случае у нас имеется 10 выходных классов ($n = 10$), представляющих цифры 0–9. Для каждого класса модель определяет вероятность, что данное входное изображение принадлежит этому классу. Чтобы вычислить общую стоимость, найдем сумму (в данном случае квадратичных) стоимостей для всех классов:

$$C_0 = \sum_{n=1}^{n} \left(a_n^L - y_n \right)^2 . \qquad \text{(Б.11)}$$

В уравнении Б.11 члены a^L и y являются векторами, каждый из которых содержит n элементов.

Исследуя $\partial C / \partial w^L$ для этого выходного слоя, мы должны учесть тот факт, что в конечном скрытом слое может быть много нейронов, каждый из которых связан с каждым нейроном в выходном слое. Здесь полезно немного поменять нотацию: обозначим последний скрытый слой как i, а выходной слой — как j. В результате получаем матрицу весов, строки которой представляют нейроны выходного слоя, а столбцы — нейроны скрытого слоя, и каждый вес можно обозначить как w_{ji}. Теперь найдем градиент для каждого веса (напомню, что у нас имеется $i \times j$ весов: по одному для каждой связи между каждой парой нейронов в двух слоях):

$$\frac{\partial C}{\partial w_{ji}^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{ji}^L} . \qquad \text{(Б.12)}$$

Эти вычисления выполняются для каждого веса в слое, и в результате получается вектор градиентов весов длиной $i \times j$.

По сути это все то же уравнение обратного распространения с единственным нейроном в слое (см. уравнение Б.7), но вот уравнение градиента стоимости относительно выхода предыдущего слоя a_{L-1} (то есть определяющее значение δ_{L-1}) теперь выглядит иначе. Поскольку этот градиент состоит из частных про-

изводных нескольких входов и весов текущего слоя, мы должны все сложить. Если придерживаться обозначений *i* и *j*, то получается выражение:

$$\delta_{L-1} = \frac{\partial C}{\partial a_i^{L-1}} = \sum_{j=0}^{n_j-1} \frac{\partial C}{\partial a_j^{L-1}} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial a_i^{L-1}}. \quad \text{(Б.13)}$$

Это уравнение выглядит довольно сложным, поэтому попробуем немного упростить его: по сравнению с Б.1—Б.10 для более простой сети все последующие уравнения почти не изменились, разве что вместо градиента для одного веса в нем вычисляется градиент для нескольких весов (уравнение Б.12). Чтобы вычислить градиент для любого заданного веса, нужно значение δ, которое само состоит из ошибок по числу связей с предыдущим слоем, поэтому мы вычисляем сумму всех этих ошибок (уравнение Б.13).

В

PYTORCH

В этом приложении мы познакомим вас с отличительными особенностями библиотеки PyTorch, а затем сравним ее с основным конкурентом — TensorFlow.

ОСОБЕННОСТИ PYTORCH

В главе 14 мы дали общий обзор библиотеки PyTorch. В этом разделе мы продолжим изучение основных ее особенностей.

СИСТЕМА AUTOGRAD

В своей работе библиотека PyTorch использует так называемую *систему Autograd*, основанную на принципе автоматического дифференцирования в обратном режиме. Как подробно описано в главе 7, конечным продуктом прямого распространения в глубокой нейронной сети являются результаты ряда связанных функций. Автоматическое дифференцирование в обратном режиме применяет цепное правило для дифференциации входов относительно стоимости в конце, действуя в обратном направлении (как рассказывается в главе 8 и в приложении Б). В каждой итерации активации нейронов в сети вычисляются на этапе прямого распространения, и каждая функция записывается в граф. В конце обучения этот граф можно раскрутить в обратном направлении, чтобы вычислить градиент для каждого нейрона.

ДИНАМИЧЕСКАЯ ИНФРАСТРУКТУРА

Что делает систему Autograd особенно интересной, так это динамический характер инфраструктуры: порядок вычислений на этапе обратного распространения определяется при каждом прямом проходе. Это важно, потому что в этом случае этап обратного распространения зависит только от того, как выполняется ваш код, то есть вычисления, выполняемые при обратном распространении, могут меняться с каждым прямым проходом. По этой причине каждый этап обучения (см. рис. 8.5) может отличаться. Это может пригодиться, например, в задачах

обработки естественного языка, когда длина входной последовательности обычно выбирается равной максимальной длине (длине самого длинного предложения в корпусе), а более короткие последовательности дополняются нулями (как мы делали это в главе 11). PyTorch, напротив, изначально поддерживает входы с динамически изменяющимися размерами, избавляя от необходимости применять усечение и дополнение.

Динамическая природа инфраструктуры также означает, что она действует не асинхронно, а это значительно упрощает отладку. Когда код генерирует ошибку, можно точно увидеть, в какой строке это произошло. Кроме того, запустив соответствующую вспомогательную функцию, так называемое оперативное выполнение, можно легко заменить традиционной моделью на основе графов, в которой графы определяются заранее, что дает преимущества в скорости и оптимизации.

PYTORCH И TENSORFLOW

Теперь у вас может возникнуть вопрос: когда предпочтительнее использовать PyTorch, а когда TensorFlow? На него не существует однозначного ответа, но далее мы рассмотрим некоторые преимущества и недостатки каждой библиотеки.

Одним из важнейших критериев является популярность: в настоящее время TensorFlow используется более широко, чем PyTorch. Первая общедоступная версия PyTorch появилась в январе 2017 года, тогда как TensorFlow была выпущена за год с небольшим до этого — в ноябре 2015-го. В быстро развивающемся мире глубокого обучения это большой срок. В действительности версия PyTorch 1.0.0 была выпущена только 7 декабря 2018 года. К тому времени TensorFlow уже пользовалась значительной популярностью, и в Интернете появилось большое количество учебных пособий и сообщений на Stack Overflow, что дало библиотеке от Google преимущество.

Второе важное обстоятельство — динамический интерфейс PyTorch, который существенно упрощает и ускоряет итерации по сравнению с TensorFlow, имеющей статическую природу[1]. PyTorch позволяет определять, изменять и выполнять узлы по мере продвижения, тогда как TensorFlow требует заранее определить архитектуру модели целиком. Отладка с PyTorch значительно проще в основном потому, что графы определяются во время выполнения. Это означает, что ошибки возникают при выполнении кода, поэтому их легко локализовать.

Визуализация в TensorFlow проста и понятна благодаря встроенной платформе TensorBoard (см. рис. 9.8). Однако в настоящее время уже появилась интеграция TensorBoard с PyTorch, а кроме того, имеются косвенные способы получения информации о том, как протекает обучение моделей PyTorch, что открывает

[1] Оперативный режим, поддержка которого появилась в TensorFlow 2.0, должен устранить этот недостаток.

возможности для разработки своих решений с использованием других библиотек (например, matplotlib).

Библиотека TensorFlow используется компанией Google в разработке и в промышленной эксплуатации, поэтому она поддерживает гораздо более сложные варианты развертывания, в том числе на мобильных устройствах и в распределенных окружениях. PyTorch исторически отставала в этих сферах; однако с выходом версии PyTorch 1.0.0, устранившей эти недостатки, стал доступен новый JIT-компилятор и новая библиотека поддержки распределенных вычислений. Кроме того, все крупные облачные провайдеры объявили об интеграции PyTorch с поддержкой TensorBoard и TPU в Google Cloud![1]

С точки зрения повседневного использования PyTorch выглядит более родной для Python, чем TensorFlow: она изначально создавалась как библиотека для Python и поэтому выглядит более простой и понятной для разработчиков на Python. Библиотека TensorFlow, напротив, даже при том, что имеет широко распространенную реализацию на Python, изначально была написана на C++, поэтому ее реализация на Python может показаться громоздкой и нескладной. Конечно, Keras отчасти решает эту проблему, но при этом она делает недоступными некоторые возможности TensorFlow[2]. Что касается Keras, PyTorch включает библиотеку Fast.ai[3], реализующую высокоуровневые абстракции для PyTorch, подобные тем, что предлагает Keras для TensorFlow.

Учитывая все вышесказанное, если вы активно занимаетесь исследованиями или ваши требования к окружению времени выполнения не очень высоки, тогда библиотека PyTorch может быть оптимальным выбором. Скорость итераций во время проведения экспериментов в сочетании с простотой отладки и обширной интеграцией с NumPy делают эту библиотеку отличным инструментом для исследований. Но если вы создаете модели глубокого обучения для промышленной эксплуатации, предпочтительнее использовать TensorFlow. Это особенно актуально для тех, кому требуется проводить обучение в распределенном окружении или использовать обученные модели на мобильной платформе.

ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ PYTORCH

В этом разделе мы рассмотрим основы установки и использования PyTorch.

---

[1] Несмотря на ожидание многих, что Google будет затягивать интеграцию с библиотекой одного из своих основных конкурентов — в данном случае Facebook.

[2] Тесная интеграция TensorFlow 2.0 с Keras призвана исправить многие из этих недостатков.

[3] github.com/fastai/fastai

УСТАНОВКА PYTORCH

Библиотека PyTorch, помимо TensorFlow и Keras, входит в состав контейнера Docker, который мы рекомендовали установить[1] для запуска блокнотов Jupyter с примерами для этой книги. То есть если вы последовали нашим рекомендациям, тогда у вас уже есть все необходимое. Если вы используете свое окружение, ознакомьтесь с инструкциями по установке, доступными на домашней странице PyTorch[2].

ОСНОВНЫЕ КОМПОНЕНТЫ PYTORCH

Фундаментальными компонентами в PyTorch являются тензоры и переменные, которые мы опишем ниже.

Базовые операции с тензорами

Как и в TensorFlow, *тензорами* здесь называются самые обычные матрицы или векторы. Функционально тензоры подобны массивам NumPy, за исключением того, что PyTorch предлагает специальные методы для вычислений с ними на графических процессорах. Под капотом эти тензоры также хранят графы (для системы autograd) и градиенты.

По умолчанию в роли тензора используется тип `FloatTensor`. В PyTorch поддерживается восемь *типов* тензоров, которые могут содержать целые или вещественные числа. При выборе типа тензора часто учитываются объем памяти, занимаемый им, и точность представления его элементов; 8-разрядные целые числа могут иметь всего 256 значений ([0 : 255]) и занимают намного меньше памяти, чем 64-разрядные[3]. Однако если целых чисел от 0 до 255 более чем достаточно, нет смысла использовать целочисленные типы, способные представлять более широкий диапазон значений. Это соображение особенно актуально, когда для обучения и использования моделей предполагается использовать графические процессоры, потому что память обычно является слабым местом графических процессоров по сравнению с обычными, где увеличение объема оперативной памяти обходится относительно недорого.

```
import torch
x = torch.zeros(28, 28, 1, dtype=torch.uint8)
y = torch.randn(28, 28, 1, dtype=torch.float32)
```

Этот код (доступен в нашем блокноте *Jupyter pytorch.ipynb* вместе с другими примерами из этого приложения) создает тензор `x` с размерами 28×28×1, запол-

[1] Инструкции по установке приводятся в начале главы 5.

[2] pytorch.org

[3] 64-разрядные целые числа могут хранить значения до 263 – 1, или 9.2 квинтиллиона.

ненный нулями — значениями типа `uint8`[1]. Также можно было бы использовать `torch.ones()`, чтобы создать аналогичный тензор, заполненный единицами. Второй тензор, `y`, содержит случайные числа из стандартного нормального распределения[2]. По определению они не могут быть 8-разрядными целыми числами, поэтому мы указали тип `float32` — 32-разрядные вещественные числа.

Как уже упоминалось, тензоры имеют много общего с *n*-мерными массивами в NumPy. Например, из массива NumPy легко можно создать тензор PyTorch с помощью метода `torch.from_numpy()`. Библиотека PyTorch также реализует множество эффективных математических операций с тензорами, многие из которых имеют аналоги в NumPy.

Автоматическая дифференциация

Тензоры PyTorch изначально способны хранить вычислительный граф для сети и градиенты. Эта их способность включается установкой аргумента `require_grad` в значение `True` при создании тензора. После этого тензор получает атрибут `grad`, хранящий градиент. Первоначально он имеет значение `None`, пока не будет вызван метод `backward()` тензора. Метод `backward()` выполняет обход операций в обратном порядке и вычисляет градиент в каждой точке графа. После первого вызова `backward()` атрибут `grad` заполняется значениями градиентов.

В следующем блоке кода определяется простой тензор, с ним выполняются некоторые математические операции и вызывается метод `backward()`, чтобы выполнить обход графа в обратном порядке и вычислить градиенты. Далее атрибут `grad` будет хранить градиенты.

```
import torch

x = torch.zeros(3, 3, dtype=torch.float32, requires_grad=True)

y = x - 4
z = y**3 * 6
out = z.mean()

out.backward()

print(x.grad)
```

Поскольку при создании `x` был установлен флаг `require_grad`, мы можем выполнить обратное распространение для этой серии вычислений. PyTorch запоминает функции, участвовавшие в вычислении окончательного результата,

---

[1] Буква «u» в имени типа `uint8` означает *unsigned* (без знака), то есть этот тип представляет 8-битные целые числа из диапазона от 0 до 255, а не от –128 до 127.

[2] Стандартное нормальное распределение имеет среднее значение 0 и стандартное отклонение 1.

используя свою систему autograd, поэтому вызов `out.backward()` вычислит градиенты и сохранит их в `x.grad`. Последняя строка выведет следующее:

```
tensor([[32., 32., 32.],
        [32., 32., 32.],
        [32., 32., 32.]])
```

Как показывает этот пример, поддержка автоматической дифференциации в PyTorch избавляет от лишних хлопот. Далее мы рассмотрим основы конструирования нейронных сетей в PyTorch.

КОНСТРУИРОВАНИЕ ГЛУБОКИХ НЕЙРОННЫХ СЕТЕЙ В PYTORCH

Основная парадигма создания нейронных сетей уже знакома вам: они конструируются из нескольких слоев, объединенных в сеть (как на рис. 4.2). На протяжении всей этой книги мы использовали высокоуровневые абстракции из библиотеки Keras для низкоуровневых функций TensorFlow. Аналогично модуль `nn` в библиотеке PyTorch содержит реализации слоев, которые принимают и возвращают тензоры. В следующем примере создается сеть с двумя слоями, подобная той, которую мы использовали для классификации рукописных цифр во второй части книги:

```
import torch

# Создать тензоры со случайными значениями для входных и выходных данных
x = torch.randn(32, 784, requires_grad=True)
y = torch.randint(low=0, high=10, size=(32,))

# Определить модель с использованием класса Sequential
model = torch.nn.Sequential(
    torch.nn.linear(784, 100),
    torch.nn.Sigmoid(),
    torch.nn.Linear(100, 10),
    torch.nn.LogSoftmax(dim=1)
)

# Определить оптимизатор и функцию потерь
optimizer = torch.optim.Adam(model.parameters())
loss_fn = torch.nn.NLLLoss()

for step in range(1000):
    # Получить прогноз, выполнив прямое распространение
    y_hat = model(x)

    # Вычислить потери
    loss = loss_fn(y_hat, y)

    # Обнулить градиенты перед обратным распространением
    optimizer.zero_grad()
```

```
# Вычислить градиенты относительно потерь
loss.backward()

# Вывести результаты
print('Step: :4d - loss: :0.4f'.format(step+1, loss.item()))

# Скорректировать параметры модели
optimizer.step()
```

Рассмотрим подробно этот пример:

- Тензоры `x` и `y` используются для входных и выходных данных модели.
- С помощью класса `Sequential` мы создаем модель, состоящую из последовательности слоев (от `linear()` до `LogSoftmax()`), почти так же, как это делали при использовании Keras.
- Инициализируем оптимизатор; в данном случае используется Adam, инициализированный значениями по умолчанию. Мы также передаем оптимизатору все тензоры, которые хотели бы оптимизировать — в данном случае `model.parameters()`.
- Инициализируем функцию потерь — она не требует никаких параметров. Мы выбрали встроенную функцию потерь отрицательного логарифмического правдоподобия `torch.nn.NLLLoss()`[1].
- Вручную выполняем желаемое число циклов обучения (см. рис. 8.5) — в данном случае 1000 — и в каждом цикле:
  - вычисляем выходы модели в строке `y_hat = model(x)`;
  - вычисляем потери, используя функцию, которую определили ранее, передавая предсказанные $\hat{y}$ и истинные значения y;
  - обнуляем градиенты. Это необходимо, потому что градиенты накапливаются в буферах, а не затираются;
  - выполняем обратное распространение, чтобы вычислить градиенты с учетом потерь;
  - в заключение используем оптимизатор. Он корректирует веса модели, используя градиенты.

[1] Комбинация выходного слоя `LogSoftmax()` с функцией стоимости `torch.nn.NLLLoss()` в PyTorch эквивалентна использованию выходного слоя softmax с функцией стоимости на основе перекрестной энтропии в Keras. В PyTorch есть функция стоимости `cross_entropy()`, но она включает вычисление softmax, поэтому, если вы решите ее использовать, вам не нужно применять функцию активации softmax к выходным данным модели.

Эта процедура отличается от метода `model.fit()`, который мы использовали в Keras, и тем не менее, учитывая теорию, описанную в этой книге, и практические примеры, которые мы проработали вместе, для вас не составит труда понять, что происходит в этом фрагменте кода. Вы без особых усилий сможете адаптировать модели глубокого обучения из этой книги для использования PyTorch[1].

[1] Обратите внимание, что наш пример нейронной сети на основе PyTorch в этом приложении не учит чему-либо значимому. Потери уменьшаются, но модель просто запоминает обучающие данные, сгенерированные случайным образом (переобучается). Мы вводим случайные числа и отображаем их в другие случайные числа. Если бы мы так же случайно сгенерировали проверочные данные, потери при проверке не уменьшались бы. Если вам любопытно, можете попробовать инициализировать `x` и `y` фактическими данными, скажем, из набора MNIST (можно импортировать эти данные с помощью Keras, как показано в листинге 5.2) и обучить модель PyTorch отображать значимые отношения!

Джон Крон, Грант Бейлевельд, Аглаэ Бассенс

Глубокое обучение в картинках. Визуальный гид по искусственному интеллекту

Перевел с английского *А. Киселев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Муханова*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Г. Шкатова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 15.04.20. Формат 70x100/16. Бумага офсетная. Усл. п. л. 32,250. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87