

# ГЛУБОКОЕ ОБУЧЕНИЕ на Python

Франсуа Шолле



MANNING



# *Deep Learning with Python*

FRANÇOIS CHOLLET

M  
MANNING  
SHELTER ISLAND

Франсуа Шолле

# Глубокое обучение на Python



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

2018

ББК 32.973.2-018.1

УДК 004.43

Ш78

## Шолле Франсуа

Ш78 Глубокое обучение на Python. — СПб.: Питер, 2018. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0770-4

Глубокое обучение — Deep learning — это набор алгоритмов машинного обучения, которые моделируют высокоуровневые абстракции в данных, используя архитектуры, состоящие из множества нелинейных преобразований. Согласитесь, эта фраза звучит угрожающе. Но всё не так страшно, если о глубоком обучении рассказывает Франсуа Шолле, который создал Keras — самую мощную библиотеку для работы с нейронными сетями. Познакомьтесь с глубоким обучением на практических примерах из самых разнообразных областей. Книга делится на две части: в первой даны теоретические основы, вторая посвящена решению конкретных задач. Это позволит вам не только разобраться в основах DL, но и научиться использовать новые возможности на практике.

Обучение — это путешествие длиной в жизнь, особенно в области искусственного интеллекта, где неизвестностей гораздо больше, чем определенности.

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

ББК 32.973.2-018.1

УДК 004.43

Права на издание получены по соглашению с Manning. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617294433 англ.

ISBN 978-5-4461-0770-4

© 2018 by Manning Publications Co. All rights reserved

© Перевод на русский язык ООО Издательство «Питер», 2018

© Издание на русском языке, оформление ООО Издательство «Питер», 2018

© Серия «Библиотека программиста», 2018

# Краткое содержание

<b>ЧАСТЬ I. ОСНОВЫ ГЛУБОКОГО ОБУЧЕНИЯ.....</b>	<b>25</b>
<b>Глава 1.</b> Что такое глубокое обучение? .....	26
<b>Глава 2.</b> Прежде чем начать: математические основы нейронных сетей ..	51
<b>Глава 3.</b> Начало работы с нейронными сетями .....	81
<b>Глава 4.</b> Основы машинного обучения .....	120
<b>ЧАСТЬ II. ГЛУБОКОЕ ОБУЧЕНИЕ НА ПРАКТИКЕ .....</b>	<b>147</b>
<b>Глава 5.</b> Глубокое обучение в технологиях компьютерного зрения .....	148
<b>Глава 6.</b> Глубокое обучение для текста и последовательностей .....	210
<b>Глава 7.</b> Лучшие практики глубокого обучения продвинутого уровня ..	270
<b>Глава 8.</b> Генеративное глубокое обучение .....	307
<b>Глава 9.</b> Заключение .....	357
<b>Приложение А.</b> Установка Keras и его зависимостей в Ubuntu .....	384
<b>Приложение В.</b> Запуск Jupyter Notebook на экземпляре EC2 GPU .....	389

# Оглавление

<b>Предисловие .....</b>	<b>16</b>
<b>Благодарности .....</b>	<b>17</b>
<b>Об этой книге .....</b>	<b>19</b>
Кому адресована эта книга .....	19
Содержание книги .....	20
Требования к программному/аппаратному обеспечению .....	21
Исходный код .....	22
Форум книги .....	22
От издательства .....	22
<b>Об авторе .....</b>	<b>23</b>
<b>Об иллюстрации на обложке .....</b>	<b>24</b>
<b>ЧАСТЬ I. ОСНОВЫ ГЛУБОКОГО ОБУЧЕНИЯ .....</b>	<b>25</b>
<b>Глава 1. Что такое глубокое обучение? .....</b>	<b>26</b>
1.1. Искусственный интеллект, машинное и глубокое обучение .....	27
1.1.1. Искусственный интеллект .....	27
1.1.2. Машинное обучение .....	28
1.1.3. Обучение представлению данных .....	29
1.1.4. «Глубина» глубокого обучения .....	31
1.1.5. Принцип действия глубокого обучения в трех картинках .....	33
1.1.6. Какой ступени развития достигло глубокое обучение .....	35

1.1.7. Не верьте рекламе . . . . .	36
1.1.8. Перспективы ИИ. . . . .	37
1.2. Что было до глубокого обучения: краткая история машинного обучения . . . . .	38
1.2.1. Вероятностное моделирование . . . . .	38
1.2.2. Первые нейронные сети . . . . .	39
1.2.3. Ядерные методы. . . . .	39
1.2.4. Деревья решений, случайные леса и градиентный бустинг . . . . .	41
1.2.5. Назад к нейронным сетям. . . . .	42
1.2.6. Отличительные черты глубокого обучения . . . . .	43
1.2.7. Современный ландшафт машинного обучения. . . . .	44
1.3. Почему глубокое обучение? Почему сейчас? . . . . .	45
1.3.1. Оборудование . . . . .	45
1.3.2. Данные . . . . .	46
1.3.3. Алгоритмы . . . . .	47
1.3.4. Новая волна инвестиций. . . . .	48
1.3.5. Демократизация глубокого обучения . . . . .	48
1.3.6. Ждать ли продолжения этой тенденции? . . . . .	49
<b>Глава 2. Прежде чем начать: математические основы нейронных сетей . . . . .</b>	<b>51</b>
2.1. Первое знакомство с нейронной сетью . . . . .	51
2.2. Представление данных для нейронных сетей . . . . .	55
2.2.1. Скаляры (тензоры нулевого ранга) . . . . .	56
2.2.2. Векторы (тензоры первого ранга) . . . . .	56
2.2.3. Матрицы (тензоры второго ранга). . . . .	56
2.2.4. Тензоры третьего и высшего рангов . . . . .	57
2.2.5. Ключевые атрибуты . . . . .	57
2.2.6. Манипулирование тензорами с помощью Numpy . . . . .	59
2.2.7. Пакеты данных . . . . .	59
2.2.8. Практические примеры тензоров с данными . . . . .	60
2.2.9. Векторные данные . . . . .	60
2.2.10. Временные ряды или последовательности. . . . .	61

2.2.11. Изображения . . . . .	61
2.2.12. Видео . . . . .	62
2.3. Шестеренки нейронных сетей: операции с тензорами . . . . .	63
2.3.1. Поэлементные операции. . . . .	63
2.3.2. Расширение . . . . .	64
2.3.3. Скалярное произведение тензоров . . . . .	65
2.3.4. Изменение формы тензора . . . . .	68
2.3.5. Геометрическая интерпретация операций с тензорами . . . . .	69
2.3.6. Геометрическая интерпретация глубокого обучения . . . . .	70
2.4. Механизм нейронных сетей: оптимизация на основе градиента . . . . .	71
2.4.1. Что такое производная? . . . . .	72
2.4.2. Производная операций с тензорами: градиент . . . . .	73
2.4.3. Стохастический градиентный спуск. . . . .	74
2.4.4. Объединение производных: алгоритм обратного распространения ошибки. . . . .	77
2.5. Оглядываясь на первый пример . . . . .	78
Краткие итоги главы . . . . .	79

**Глава 3. Начало работы с нейронными сетями . . . . .** **81**

3.1. Анатомия нейронной сети . . . . .	82
3.1.1. Слои: строительные блоки глубокого обучения. . . . .	83
3.1.2. Модели: сети слоев . . . . .	84
3.1.3. Функции потерь и оптимизаторы: ключи к настройке процесса обучения . . . . .	84
3.2. Введение в Keras . . . . .	85
3.2.1. Keras, TensorFlow, Theano и CNTK . . . . .	86
3.2.2. Разработка с использованием Keras: краткий обзор. . . . .	87
3.3. Настройка рабочей станции для глубокого обучения . . . . .	89
3.3.1. Jupyter Notebook: предпочтительный способ проведения экспериментов с глубоким обучением . . . . .	90
3.3.2. Подготовка Keras: два варианта . . . . .	90
3.3.3. Запуск заданий глубокого обучения в облаке: за и против . . . . .	91
3.3.4. Выбор GPU для глубокого обучения . . . . .	91

3.4. Классификация отзывов к фильмам: пример бинарной классификации . . . . .	92
3.4.1. The IMDB dataset . . . . .	92
3.4.2. Подготовка данных . . . . .	93
3.4.3. Конструирование сети . . . . .	94
3.4.4. Проверка решения . . . . .	98
3.4.5. Использование обученной сети для предсказаний на новых данных . . . . .	101
3.4.6. Дальнейшие эксперименты . . . . .	101
3.4.7. Подведение итогов . . . . .	102
3.5. Классификация новостных лент: пример классификации в несколько классов . . . . .	102
3.5.1. Набор данных Reuters . . . . .	103
3.5.2. Подготовка данных . . . . .	104
3.5.3. Конструирование сети . . . . .	105
3.5.4. Проверка решения . . . . .	106
3.5.5. Предсказания на новых данных . . . . .	108
3.5.6. Другой способ обработки меток и потерю . . . . .	109
3.5.7. Важность использования достаточно больших промежуточных слоев . . . . .	109
3.5.8. Дальнейшие эксперименты . . . . .	110
3.5.9. Подведение итогов . . . . .	110
3.6. Предсказание цен на дома: пример регрессии . . . . .	111
3.6.1. Набор данных с ценами на жилье в Бостоне . . . . .	111
3.6.2. Подготовка данных . . . . .	112
3.6.3. Конструирование сети . . . . .	112
3.6.4. Оценка решения методом перекрестной проверки по K блокам . . . . .	113
3.6.5. Подведение итогов . . . . .	118
Краткие итоги главы . . . . .	118
<b>Глава 4. Основы машинного обучения . . . . .</b>	<b>120</b>
4.1. Четыре раздела машинного обучения . . . . .	120
4.1.1. Контролируемое обучение . . . . .	121
4.1.2. Неконтролируемое обучение . . . . .	121

4.1.3. Самоконтролируемое обучение . . . . .	122
4.1.4. Обучение с подкреплением . . . . .	122
4.2. Оценка моделей машинного обучения . . . . .	124
4.2.1. Тренировочные, проверочные и контрольные наборы данных . . . . .	124
4.2.2. Что важно помнить . . . . .	128
4.3. Обработка данных, конструирование признаков и обучение признаков . . . . .	128
4.3.1. Предварительная обработка данных для нейронных сетей . . . . .	129
4.3.2. Конструирование признаков . . . . .	130
4.4. Переобучение и недообучение . . . . .	132
4.4.1. Уменьшение размера сети . . . . .	133
4.4.2. Добавление регуляризации весов . . . . .	136
4.4.3. Добавление прореживания . . . . .	138
4.5. Обобщенный процесс решения задач машинного обучения . . . . .	140
4.5.1. Определение задачи и создание набора данных . . . . .	140
4.5.2. Выбор меры успеха . . . . .	141
4.5.3. Выбор протокола оценки . . . . .	142
4.5.4. Предварительная подготовка данных . . . . .	142
4.5.5. Разработка модели, более совершенной, чем базовый случай . . . . .	143
4.5.6. Масштабирование по вертикали: разработка модели с переобучением . . . . .	144
4.5.7. Регуляризация модели и настройка гиперпараметров . . . . .	145
Краткие итоги главы . . . . .	146

**ЧАСТЬ II. ГЛУБОКОЕ ОБУЧЕНИЕ НА ПРАКТИКЕ . . . . .** **147**

<b>Глава 5. Глубокое обучение в технологиях компьютерного зрения . . . . .</b>	<b>148</b>
5.1. Введение в сверточные нейронные сети . . . . .	148
5.1.1. Операция свертывания . . . . .	151
5.1.2. Выбор максимального значения из соседних (max-pooling) . . . . .	157
5.2. Обучение сверточной нейронной сети с нуля на небольшом наборе данных . . . . .	159
5.2.1. Целесообразность глубокого обучения для решения задач с небольшими наборами данных . . . . .	159

---

5.2.2. Загрузка данных . . . . .	160
5.2.3. Конструирование сети . . . . .	163
5.2.4. Предварительная обработка данных . . . . .	165
5.2.5. Расширение данных . . . . .	169
5.3. Использование предварительно обученной сверточной нейронной сети . . . . .	173
5.3.1. Выделение признаков . . . . .	174
5.3.2. Дообучение . . . . .	184
5.3.3. Подведение итогов . . . . .	190
5.4. Визуализация знаний, заключенных в сверточной нейронной сети . . . . .	191
5.4.1. Визуализация промежуточных активаций . . . . .	191
5.4.2. Визуализация фильтров сверточных нейронных сетей . . . . .	198
5.4.3. Визуализация тепловых карт активации класса . . . . .	204
Краткие итоги главы . . . . .	209

## **Глава 6. Глубокое обучение для текста и последовательностей . . . 210**

6.1. Работа с текстовыми данными . . . . .	211
6.1.1. Прямое кодирование слов и символов . . . . .	213
6.1.2. Векторное представление слов . . . . .	215
6.1.3. Объединение всего вместе: от исходного текста к векторному представлению слов . . . . .	221
6.1.4. Подведение итогов . . . . .	228
6.2. Рекуррентные нейронные сети . . . . .	228
6.2.1. Рекуррентный слой в Keras . . . . .	231
6.2.2. Слои LSTM и GRU . . . . .	235
6.2.3. Пример использования слоя LSTM из Keras . . . . .	238
6.2.4. Подведение итогов . . . . .	240
6.3. Улучшенные методы использования рекуррентных нейронных сетей . . . . .	240
6.3.1. Задача прогнозирования температуры . . . . .	241
6.3.2. Подготовка данных . . . . .	244
6.3.3. Базовое решение без привлечения машинного обучения . . . . .	247
6.3.4. Базовое решение с привлечением машинного обучения . . . . .	248
6.3.5. Первое базовое рекуррентное решение . . . . .	250

6.3.6. Использование рекуррентного прореживания для борьбы с переобучением . . . . .	251
6.3.7. Наложение нескольких рекуррентных слоев друг на друга . . . . .	253
6.3.8. Использование двунаправленных рекуррентных нейронных сетей . . . . .	254
6.3.9. Что дальше . . . . .	258
6.3.10. Подведение итогов . . . . .	259
6.4. Обработка последовательностей с помощью сверточных нейронных сетей . . . . .	260
6.4.1. Обработка последовательных данных с помощью одномерной сверточной нейронной сети . . . . .	260
6.4.2. Выбор соседних значений в одномерной последовательности данных . . . . .	261
6.4.3. Реализация одномерной сверточной сети . . . . .	262
6.4.4. Объединение сверточных и рекуррентных сетей для обработки длинных последовательностей . . . . .	264
6.4.5. Подведение итогов . . . . .	268
Краткие итоги главы . . . . .	268

## **Глава 7. Лучшие практики глубокого обучения продвинутого уровня . . . . .** **270**

7.1. За рамками модели Sequential: функциональный API фреймворка Keras . . . . .	270
7.1.1. Введение в функциональный API . . . . .	274
7.1.2. Модели с несколькими входами . . . . .	275
7.1.3. Модели с несколькими выходами . . . . .	277
7.1.4. Ориентированные ациклические графы уровней . . . . .	280
7.1.5. Повторное использование экземпляров слоев . . . . .	284
7.1.6. Модели как слои . . . . .	285
7.1.7. Подведение итогов . . . . .	286
7.2. Исследование и мониторинг моделей глубокого обучения с использованием обратных вызовов Keras и TensorBoard . . . . .	286
7.2.1. Применение обратных вызовов для воздействия на модель в ходе обучения . . . . .	287
7.2.2. Введение в TensorBoard: фреймворк визуализации TensorFlow . . . . .	290
7.2.3. Подведение итогов . . . . .	296

---

7.3. Извлечение максимальной пользы из моделей . . . . .	297
7.3.1. Шаблоны улучшенных архитектур. . . . .	297
7.3.2. Оптимизация гиперпараметров . . . . .	301
7.3.3. Ансамблирование моделей . . . . .	303
7.3.4. Подведение итогов . . . . .	305
Краткие итоги главы . . . . .	306
<b>Глава 8. Генеративное глубокое обучение . . . . .</b>	<b>307</b>
8.1. Генерирование текста с помощью LSTM . . . . .	309
8.1.1. Краткая история генеративных рекуррентных сетей . . . . .	309
8.1.2. Как генерируются последовательности данных? . . . . .	310
8.1.3. Важность стратегии выбора . . . . .	311
8.1.4. Реализация посимвольной генерации текста на основе LSTM . . . . .	313
8.1.5. Подведение итогов . . . . .	318
8.2. DeepDream . . . . .	318
8.2.1. Реализация DeepDream в Keras . . . . .	320
8.2.2. Подведение итогов . . . . .	326
8.3. Нейронная передача стиля . . . . .	326
8.3.1. Функция потерь содержимого . . . . .	327
8.3.2. Функция потерь стиля . . . . .	328
8.3.3. Нейронная передача стиля в Keras . . . . .	329
8.3.4. Подведение итогов . . . . .	336
8.4. Генерирование изображений с вариационными автокодировщиками . . . . .	336
8.4.1. Выбор шаблонов из скрытых пространств изображений . . . . .	336
8.4.2. Концептуальные векторы для редактирования изображений . . . . .	338
8.4.3. Вариационные автокодировщики . . . . .	339
8.4.4. Подведение итогов . . . . .	346
8.5. Введение в генеративно-состязательные сети . . . . .	346
8.5.1. Реализация простейшей генеративно-состязательной сети . . . . .	348
8.5.2. Набор хитростей . . . . .	349
8.5.3. Генератор . . . . .	350
8.5.4. Дискриминатор . . . . .	351

8.5.5. Состязательная сеть . . . . .	352
8.5.6. Как обучить сеть DCGAN . . . . .	353
8.5.7. Подведение итогов . . . . .	355
Краткие итоги главы . . . . .	356
<b>Глава 9. Заключение . . . . .</b>	<b>357</b>
9.1. Краткий обзор ключевых понятий . . . . .	358
9.1.1. Разные подходы к ИИ . . . . .	358
9.1.2. Что делает глубокое обучение особенным среди других подходов к машинному обучению . . . . .	358
9.1.3. Как правильно воспринимать глубокое обучение . . . . .	359
9.1.4. Ключевые технологии . . . . .	360
9.1.5. Обобщенный процесс машинного обучения . . . . .	361
9.1.6. Основные архитектуры сетей . . . . .	362
9.1.7. Пространство возможностей . . . . .	367
9.2. Ограничения глубокого обучения . . . . .	369
9.2.1. Риск очеловечивания моделей глубокого обучения . . . . .	370
9.2.2. Локальное и экстремальное обобщение . . . . .	372
9.2.3. Подведение итогов . . . . .	373
9.3. Будущее глубокого обучения . . . . .	374
9.3.1. Модели как программы . . . . .	375
9.3.2. За границами алгоритма обратного распространения ошибки и дифференцируемых слоев . . . . .	377
9.3.3. Автоматизированное машинное обучение . . . . .	377
9.3.4. Непрерывное обучение и повторное использование модульных подпрограмм . . . . .	378
9.3.5. Долгосрочная перспектива . . . . .	380
9.4. Как не отстать от прогресса в быстро развивающейся области . . . . .	381
9.4.1. Практические решения реальных задач на сайте Kaggle . . . . .	381
9.4.2. Знакомство с последними разработками на сайте arXiv . . . . .	382
9.4.3. Исследование экосистемы Keras . . . . .	383
9.5. Заключительное слово . . . . .	383

**Приложение А. Установка Keras и его зависимостей в Ubuntu . . . . . 384**

A.1. Установка пакетов научных вычислений для Python . . . . .	385
A.2. Настройка поддержки GPU . . . . .	386
A.3. Установка Theano (необязательно) . . . . .	387
A.4. Установка Keras . . . . .	388

**Приложение В. Запуск Jupyter Notebook на экземпляре EC2 GPU 389**

B.1. Что такое Jupyter Notebook? Зачем запускать Jupyter Notebook на AWS GPU? . . . . .	389
B.2. Когда нежелательно использовать Jupyter на AWS для глубокого обучения? . . . . .	390
B.3. Настройка экземпляра AWS GPU . . . . .	390
B.3.1. Настройка Jupyter. . . . .	393
B.4. Установка Keras . . . . .	394
B.5. Настройка перенаправления локальных портов . . . . .	395
B.6. Доступ к Jupyter из браузера на локальном компьютере . . . . .	395

Книга Франсуа Шолле "Глубокое обучение на Python" была куплена в рескладчину за 10 рублей на сайте опен хайд биз

# Предисловие

Если вы выбрали эту книгу, то вы, вероятно, наслышаны о недавнем небывалом успехе методики глубокого обучения в области искусственного интеллекта (ИИ). Всего за пять лет мы прошли путь от почти никуда не годного распознавания образов и речи до невероятно эффективного решения этих задач.

Последствия такого внезапного прогресса отразились почти на всех отраслях. Однако для того, чтобы начать внедрение технологии глубокого обучения во все задачи, которые можно решить с ее применением, мы должны сделать ее доступной как можно большему количеству людей, включая неспециалистов, то есть тех, кто не является инженером-исследователем или аспирантом. Чтобы раскрыть весь потенциал глубокого обучения, мы должны полностью демократизировать его.

Когда в марте 2015 года я выпустил первую версию Keras, фреймворка глубокого обучения, я не задумывался о демократизации искусственного интеллекта. Я несколько лет занималась исследованиями в области машинного обучения и создал Keras себе в помощь в экспериментах. Однако в течение 2015 и 2016 годов десятки тысяч новых людей открыли для себя область глубокого обучения; многие из них выбрали Keras, потому что он был — и остается — самым простым фреймворком для начинающих. Я увидел, как десятки новичков используют Keras самыми неожиданными и достаточно действенными способами, и тогда пришел к мысли, что мне нужно подумать о доступности и демократизации ИИ. Я осознал, что чем шире мы будем распространять эти технологии, тем ценнее они будут становиться. Доступность быстро стала одной из главных целей Keras, и за несколько лет сообществу разработчиков удалось добиться фантастических достижений в этом направлении. Мы в буквальном смысле «вручили» технологию глубокого обучения десяткам тысяч людей, и они, в свою очередь, воспользовались ею для решения важных проблем, о существовании которых мы до недавнего времени даже не подозревали.

Книга, которую вы держите, — еще один шаг на пути популяризации глубокого обучения. Фреймворку Keras всегда требовался сопроводительный курс, который одновременно освещал бы основы глубокого обучения, показывал примеры его использования и демонстрировал лучшие практики в применении глубокого

обучения. Эта книга — моя лучшая попытка по созданию такого курса. Я писал ее, стараясь максимально доступно объяснить идеи, лежащие в основе глубокого обучения и его реализации. Это не значит, что я преднамеренно упрощал изложение, — я всецело уверен, что в теме глубокого обучения нет ничего сложного. Надеюсь, эта книга принесет вам пользу и поможет начать создавать интеллектуальные приложения и решать важные для вас проблемы.

[openhide.biz](http://openhide.biz)

## Благодарности

Я хочу поблагодарить сообщество Keras за помощь в создании этой книги. В настоящее время в проекте Keras насчитывается несколько сотен добровольных разработчиков и более 200 000 пользователей. Ваш вклад и отзывы помогли превратить Keras в то, чем он является сейчас.

Также я хочу поблагодарить компанию Google за поддержку проекта Keras. Было очень приятно, когда в Google решили использовать Keras в качестве высокоуровневого API для TensorFlow<sup>1</sup>. Бесшовная интеграция Keras и TensorFlow выгодна пользователям обоих продуктов. Связка TensorFlow и Keras делает технологии глубокого обучения доступными для широкого круга.

Я хочу поблагодарить сотрудников издательства Manning, кто сделал возможным выпуск этой книги: издателя Марджана Бейса (Marjan Vase) и всех сотрудников редакторского и технического отделов, в том числе Кристину Тейлор (Christina Taylor), Жанет Вайл (Janet Vail), Тиффани Тейлор (Tiffany Taylor), Кэти Теннант (Katie Tennant), Дотти Марсико (Dottie Marsico) и многих других, чья работа осталась «за кадром».

Большое спасибо техническим рецензентам во главе с Александром Драгосавлевичем (Aleksandar Dragosavljević): Диего Акунье Розасу (Diego Acuña Rozas), Джиффи Барто (Geoff Barto), Дэвиду Блюменталь-Барби (David Blumenthal-Barby), Абелю Брауну (Abel Brown), Кларку Дорману (Clark Dorman), Кларку Гейлорду (Clark Gaylord), Томасу Хайману (Thomas Heiman), Уилсону Мару (Wilson Mar), Сумиту Палу (Sumit Pal), Владимиру Пасману (Vladimir Pasman), Густаво Патино (Gustavo Patino), Питеру Рабиновичу (Peter Rabinovitch), Эльвину Раджу (Alvin Raj), Клаудио Родригесу (Claudio Rodriguez), Срджану Сантичу (Srdjan Santic), Ричарду Тобиасу (Richard Tobias), Мартину Верзилли (Martin Verzilli), Уильяму Уилеру

<sup>1</sup> Открытая программная библиотека алгоритмов машинного обучения, разработанная компанией Google для решения задач построения и тренировки нейронной сети с целью автоматического поиска и классификации образов с качеством человеческого восприятия. — Примеч. пер.

(William E. Wheeler) и Дэниэлу Уильямсу (Daniel Williams) — и всем участникам форума. Они без устали выискивали технические ошибки, ошибки в терминологии и опечатки, а также вносили предложения по темам. Каждый цикл технического рецензирования и каждый отзыв, оставленный на форуме, был учтен и повлиял на рукопись.

Особое спасибо Джерри Гейнсу (Jerry Gaines), выступившему в роли технического редактора; Алексу Отту (Alex Ott) и Ричарду Тобиасу (Richard Tobias), выполнившим техническую редактуру книги. Они — лучшие технические редакторы.

Наконец, я хочу выразить благодарность моей супруге Марии за безграничную поддержку на протяжении всего времени работы над Keras и над этой книгой.

# Об этой книге

Книга написана для всех желающих начать изучение технологии глубокого обучения с нуля или расширить свои знания. Инженеры, работающие в области машинного обучения, разработчики программного обеспечения и студенты найдут много ценного на этих страницах.

В этой книге предлагается реальное практическое исследование глубокого обучения. Мы старались избегать математических формул, предпочитая объяснять количественные понятия с помощью фрагментов кода и формировать практическое понимание основных идей машинного и глубокого обучения.

Вы увидите более 30 примеров программного кода с подробными комментариями, практическими рекомендациями и простыми обобщенными объяснениями всего, что нужно знать для использования глубокого обучения в решении конкретных задач.

В примерах используются фреймворк глубокого обучения Keras, написанный на Python, и библиотека TensorFlow в качестве внутреннего механизма. Keras — один из популярнейших и быстро развивающихся фреймворков глубокого обучения. Он часто рекомендуется как наиболее удачный инструмент для начинающих изучать глубокое обучение.

Прочитав эту книгу, вы будете четко понимать, что такое глубокое обучение, когда оно применимо и какие ограничения имеет. Вы познакомитесь со стандартным процессом интерпретации и решения задач машинного обучения и узнаете, как бороться с часто встречающимися проблемами. Вы научитесь использовать Keras для решения практических задач — от распознавания образов до обработки естественного языка: классификации изображений, временного прогнозирования, анализа эмоций, генерации изображений и текста и многого другого.

## Кому адресована эта книга

Эта книга написана для людей с опытом программирования на Python, желающих начать знакомство с темой машинного обучения с технологией глубокого обучения. Однако она также может быть полезной для других читателей:

- ❑ Если вы специалист по обработке и анализу данных, знакомый с машинным обучением, эта книга позволит вам получить достаточно полное практическое

представление о глубоком обучении — наиболее быстро развивающемся разделе машинного обучения.

- ❑ Если вы эксперт в области глубокого обучения, желающий освоить фреймворк Keras, вы найдете в этой книге лучший интенсивный курс по Keras.
- ❑ Если вы аспирант, изучающий технологии глубокого обучения в ходе формального курса, в этой книге вы найдете практическое дополнение к своей учебе, которое поможет вам лучше понять принцип действия нейросетей и познакомит с наиболее эффективными приемами.

Даже технически мыслящие люди, которые не занимаются программированием регулярно, найдут эту книгу полезной в качестве введения в базовые и продвинутые понятия глубокого обучения.

Для использования Keras вам необходимо владеть языком Python на среднем уровне. Также будет полезно знакомство с библиотекой NumPy, хотя это и не требуется. Опыт в машинном или глубоком обучении не является обязательным условием: эта книга охватывает все необходимые основы с нуля. Не требуется иметь какой-то особенной математической подготовки — вполне достаточно знания математики на уровне средней школы.

## Содержание книги

Эта книга состоит из двух частей. Если у вас нет опыта в области машинного обучения, я настоятельно рекомендую прочитать первую часть, прежде чем переходить ко второй. Мы начнем с простых примеров и постепенно будем приближаться к современным технологиям.

Первая часть — это обобщенное введение в глубокое обучение. Здесь формируется контекст: даются определения и объясняются все понятия, которые необходимо знать, чтобы приступить к изучению машинного обучения и нейросетей.

- ❑ Глава 1 формирует контекст и знакомит с основными понятиями ИИ, машинного и глубокого обучения.
- ❑ Глава 2 знакомит с базовыми идеями, необходимыми для освоения глубокого обучения: тензоры, операции с тензорами, градиентный спуск и обратное распространение ошибки обучения. В этой главе приводится также первый пример действующей нейронной сети.
- ❑ Глава 3 включает в себя все необходимое, чтобы начать работу с нейронными сетями: введение в Keras — фреймворк глубокого обучения; руководство по настройке рабочей станции; а также три основополагающих примера с подробными пояснениями. К концу этой главы вы научитесь обучать простые нейронные сети для решения задач классификации и регрессии и получите полное представление о том, что происходит за кулисами во время обучения.

- ❑ Глава 4 изучает канонический процесс машинного обучения. Здесь вы узнаете о типичных ловушках и о том, как их избежать.

Вторая часть подробно рассказывает о практическом применении глубокого обучения в распознавании образов и обработке естественного языка. Многие примеры, представленные в этой части, можно использовать как шаблоны для решения практических проблем в глубоком обучении.

- ❑ Глава 5 рассматривает ряд практических примеров распознавания образов с особым вниманием к классификации изображений.
- ❑ Глава 6 знакомит с практическими методами обработки последовательностей данных, таких как текст и временные последовательности.
- ❑ Глава 7 включает в себя продвинутые приемы создания современных моделей глубокого обучения.
- ❑ Глава 8 объясняет генеративные, или порождающие, модели: модели глубокого обучения, способные создавать изображения и текст, причем иногда результат получается на удивление художественным.
- ❑ Глава 9 посвящена закреплению всего, о чем рассказывалось в предыдущих главах, а также описанию перспектив глубокого обучения и исследованию его вероятного будущего.

## Требования к программному/аппаратному обеспечению

Во всех примерах в этой книге используется Keras (<https://keras.io>) — фреймворк глубокого обучения с открытым исходным кодом и доступный для загрузки бесплатно. Вам понадобится доступ к компьютеру с операционной системой (ОС) UNIX; фреймворк можно также использовать в Windows, но я не рекомендую это делать. Полное описание процесса настройки приводится в приложении А.

Желательно, чтобы компьютер был оснащен современной видеокартой NVIDIA, такой как TITAN X. Это необязательно, но поможет улучшить впечатления за счет ускорения примеров в несколько раз. Подробнее о настройке рабочей станции рассказывается в разделе 3.3.

Если у вас нет доступа к рабочей станции с современной видеокартой NVIDIA, можете воспользоваться облачным окружением. Например, можно использовать экземпляры Google Cloud (такие, как n1-standard-8, оснащенные графическими ускорителями NVIDIA Tesla K80) или экземпляры Amazon Web Services (AWS) с GPU (такие, как p2.xlarge). В приложении В представлен один из возможных вариантов использования экземпляра в облаке AWS посредством Jupyter Notebook из браузера.

## Исходный код

Код всех примеров в книге доступен для загрузки в виде блокнотов Jupyter Notebooks на сайте книги [www.manning.com/books/deep-learning-with-python](http://www.manning.com/books/deep-learning-with-python), а также в репозитории GitHub <https://github.com/fchollet/deep-learning-with-python-notebooks>.

## Форум книги

Покупка книги «Глубокое обучение с Python» включает бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу <https://forums.manning.com/forums/deep-learning-with-python>. Узнать больше о других форумах на сайте издательства Manning и познакомиться с правилами вы сможете на странице <https://forums.manning.com/forums/about>.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Однако со стороны автора отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание — его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору стимулирующие вопросы, чтобы его интерес не угасал!

Форум и архивы предыдущих обсуждений будут доступны на сайте издательства, пока книга находится в печати.

## От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

# Об авторе



Франсуа Шолле (François Chollet) занимается проблематикой глубокого обучения в Google, в городе Маунтин-Вью, штат Калифорния. Является создателем Keras — библиотеки глубокого обучения, а также участником проекта по разработке фреймворка машинного обучения TensorFlow. Также занимается исследованиями в области машинного обучения, основное внимание уделяя распознаванию образов и применению машинного обучения к формальным рассуждениям. Выступал с докладами на крупных конференциях в этой области, включая «Conference on Computer Vision and Pattern Recognition» (CVPR), «Conference and Workshop on Neural Information Processing Systems» (NIPS), «International Conference on Learning Representations» (ICLR) и др.

# Об иллюстрации на обложке

Иллюстрация на обложке «Глубокое обучение с Python» подписана как «Одежда персидской женщины в 1568 году». Она взята из книги «Collection of the Dresses of Different Nations, Ancient and Modern» (Коллекция костюмов разных народов, античных и современных) Томаса Джейфериса (Thomas Jefferys), опубликованной в Лондоне между 1757 и 1772 годами. На титульной странице указано, что это выполненная вручную каллиграфическая цветная гравюра, обработанная гумми-арабиком.

Томас Джейферис (1719–1771) носил звание «Географ короля Георга III». Английский картограф, он был ведущим поставщиком карт того времени. Он выгравировал и напечатал множество карт для нужд правительства, других официальных органов и широкий спектр коммерческих карт и атласов, в частности карт Северной Америки. Будучи картографом, интересовался местной одеждой народов, населяющих разные земли, и собрал блестящую коллекцию различных платьев, описав ее в четырех томах.

Очарование далеких земель и дальних путешествий для удовольствия было относительно новым явлением в конце XVIII века, и коллекции, подобные этой, были весьма популярны, так как позволяли ознакомиться с внешним видом жителей других стран. Разнообразие рисунков, собранных Джейферисом, свидетельствует о проявлении 200 лет назад яркой индивидуальности и уникальности народов мира. С тех пор стиль одежды сильно изменился и исчезло разнообразие, характеризующее различные области и страны. Теперь трудно различить по одежде даже жителей разных континентов. Если взглянуть на это с оптимистичной точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду более насыщенной личной жизни или в угоду более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну техническую книгу от другой, издательство Manning проявляет инициативу и деловую сметку, украшая обложки книг изображениями, основанными на богатом разнообразии жизненного уклада народов двухвековой давности, придав новую жизнь рисункам Джейфериса.

# ЧАСТЬ I

## Основы глубокого обучения

Главы 1–4 этой книги дадут вам базовое представление о том, что такое глубокое обучение, где оно применяется и как работает. Также вы познакомитесь с каноническим процессом решения задач анализа данных методами глубокого обучения. Если вы еще не очень хорошо ориентируетесь в глубоком обучении, обязательно прочитайте первую часть книги от начала до конца, прежде чем переходить к описанию приемов практического применения, о которых рассказывается во второй части.

# 1

# Что такое глубокое обучение?

Эта глава охватывает следующие темы:

- ✓ обобщенные определения основных понятий;
- ✓ история развития машинного обучения;
- ✓ ключевые факторы роста популярности глубокого обучения и потенциал этой дисциплины в будущем.

За последние несколько лет тема искусственного интеллекта (ИИ) вызвала большую шумиху в средствах массовой информации. Машинное обучение, глубокое обучение и ИИ упоминались в бесчисленном количестве статей, многие из которых никак не связаны с описанием технологий. Нам обещали появление виртуальных собеседников, автомобилей с автопилотом и виртуальных помощников. Иногда будущее рисовали в мрачных тонах, а иногда изображали утопическим: освобождение людей от рутинного труда и выполнение основной работы роботами, наделенными искусственным интеллектом. Будущему или настоящему специалисту в области машинного обучения важно уметь выделять полезный сигнал из шума, видеть в раздутых пресс-релизах изменения, действительно способные повлиять на мир. Наше будущее поставлено на карту, и вам предстоит сыграть в нем активную роль: закончив чтение этой книги, вы вольетесь в ряды тех, кто разрабатывает системы ИИ. Поэтому давайте рассмотрим следующие вопросы: чего уже достигло глубокое обучение? насколько это важно? в каком направление пойдет дальнейшее развитие? можно ли верить поднятой шумихе?

Эта глава закладывает фундамент для дальнейшего обсуждения ИИ, машинного и глубокого обучения.

## 1.1. Искусственный интеллект, машинное и глубокое обучение

Прежде всего определим, что подразумевается под искусственным интеллектом. Что такое ИИ, машинное и глубокое обучение (рис. 1.1)? Как они связаны друг с другом?



**Рис. 1.1.** Искусственный интеллект, машинное и глубокое обучение

### 1.1.1. Искусственный интеллект

Идея искусственного интеллекта появилась в 1950-х, когда группа энтузиастов из только зарождающейся области информатики задались вопросом, можно ли заставить компьютеры «думать», — вопросом, последствия которого мы изучаем до сих пор. Коротко эту область можно определить так: *автоматизация интеллектуальных задач, обычно выполняемых людьми*. Соответственно, ИИ — это область, охватывающая машинное обучение и глубокое обучение, а также включающая в себя многие подходы, не связанные с обучением. Например, первые программы для игры в шахматы действовали по жестко определенным правилам, заданным программистами, и не могли квалифицироваться как осуществляющие машинное обучение. Долгое время многие эксперты полагали, что ИИ уровня человека можно создать, если дать программисту достаточный набор явных правил для манипулирования знаниями. Этот поход, известный как *символический ИИ*, и являлся доминирующей парадигмой ИИ с 1950-х до конца 1980-х. Пик его популярности пришелся на *бум экспертиных систем* в 1980-х.

Символический ИИ прекрасно справлялся с решением четко определенных логических задач, таких как игра в шахматы, но, как оказалось, невозможно задать строгие правила для решения более сложных, нечетких задач, таких как классификация изображений, распознавание речи и перевод на другие языки. На смену символическому ИИ появился новый подход: *машинное обучение*.

### 1.1.2. Машинное обучение

В викторианской Англии жила леди Ада Лавлейс (Ada Lovelace) — друг и соратник Чарльза Бэббиджа (Charles Babbage), изобретателя *аналитической вычислительной машины*: первого известного механического компьютера. Несомненно, аналитическая машина опередила свое время, но она не задумывалась как универсальный компьютер, когда разрабатывалась в 1830-х и 1840-х, потому что идея универсальных вычислений еще не родилась. Эта машина просто давала возможность использовать механические операции для автоматизации некоторых вычислений из области математического анализа, что и обусловило такое ее название. В 1843 году Ада Лавлейс заметила: «Аналитическая машина не может создавать что-то новое. Она может делать все, что мы и сами знаем, как выполнять... ее цель состоит лишь в том, чтобы помогать нам осуществлять то, с чем мы уже хорошо знакомы».

Позднее пионер ИИ Алан Тьюринг (Alan Turing) в своей знаменитой статье «Computing Machinery and Intelligence»<sup>1</sup> назвал это замечание «аргументом Ады Лавлейс»<sup>2</sup>. В этой статье был представлен *тест Тьюринга*, а также перечислены основные идеи, которые могут привести к созданию ИИ. Тьюринг цитировал Аду Лавлейс, размышляя над способностью обычных компьютеров к самообучению и созданию чего-либо нового, и пришел к выводу, что да, могут.

Область машинного обучения возникла из вопроса: может ли компьютер выйти за рамки того, «что мы и сами знаем, как выполнять», и самостоятельно научиться решать некоторую определенную задачу? Может ли компьютер удивить нас? Может ли компьютер без помощи программиста, задающего правила обработки данных, автоматически определить эти правила, исследуя данные?

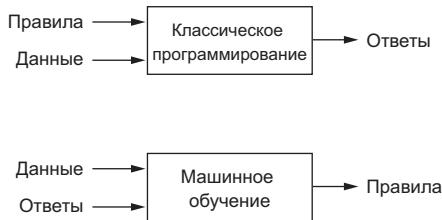
Этот вопрос открыл двери в новую парадигму программирования. В классическом программировании, в парадигме символического ИИ, люди вводят правила (программу) и данные для обработки в соответствии с этими правилами и получают ответы (рис. 1.2). В машинном обучении люди вводят данные и ответы, соответствующие этим данным, а на выходе получают правила. Эти правила затем можно применить к новым данным для получения оригинальных ответов.

В машинном обучении система *обучается*, а не программируется явно. Ей передаются многочисленные примеры, имеющие отношение к решаемой задаче, а она находит в этих примерах статистическую структуру, которая позволяет системе выработать правила для автоматического решения задачи. Например, чтобы автоматизировать задачу определения фотографий, сделанных в отпуске, можно передать системе машинного обучения множество примеров фотографий, уже классифицированных людьми, и система изучит статистические правила классификации конкретных фотографий.

---

<sup>1</sup> «Вычислительные машины и разум», перевод на русский язык можно найти по адресу: <https://bio.wikireading.ru/6066>. — Примеч. пер.

<sup>2</sup> Turing, A. M. «Computing Machinery and Intelligence», Mind 59, no. 236 (1950): 433–460.



**Рис. 1.2.** Машинное обучение: новая парадигма программирования

Расцвет машинного обучения начался только в 1990-х, но эта область быстро превратилась в наиболее популярный и успешный раздел ИИ, и эта тенденция была подкреплена появлением более быстродействующей аппаратуры и огромных наборов данных. Машинное обучение тесно связано с математической статистикой, но имеет несколько важных отличий. В отличие от статистики, машинное обучение обычно имеет дело с большими и сложными наборами данных (например, состоящими из миллионов фотографий, каждая из которых состоит из десятков тысяч пикселов), к которым практически невозможно применить классические методы статистического анализа, такие как байесовские методы. Как результат, машинное и в особенности глубокое обучение не имеют мощной математической платформы и основываются почти исключительно на инженерных решениях. Это практическая дисциплина, в которой идеи чаще доказываются эмпирически, а не теоретически.

### 1.1.3. Обучение представлению данных

Чтобы дать определение *глубокому обучению* и понять разницу между глубоким обучением и другими методами машинного обучения, сначала нужно получить некоторое представление о том, что *делают* алгоритмы машинного обучения. Чуть выше отмечалось, что машинное обучение выявляет правила решения задач обработки данных по примерам ожидаемых результатов. То есть для машинного обучения нам нужны три составляющие:

- *Контрольные входные данные* — например, если решается задача распознавания речи, такими контрольными входными данными могут быть файлы с записью речи разных людей; если решается задача классификации изображений, такими данными могут быть изображения.
- *Примеры ожидаемых результатов* — в задаче распознавания речи это могут быть транскрипции звуковых файлов, составленные людьми; в задаче классификации изображений ожидаемым результатом могут быть теги, такие как «собака», «кошка» и др.
- *Способ оценки качества работы алгоритма* — это необходимо для определения, насколько далеки отклоняются результаты, возвращаемые алгоритмом, от ожидаемых. Оценка используется как сигнал обратной связи для корректировки работы алгоритма. Этот этап корректировки мы и называем *обучением*.

Модель машинного обучения трансформирует исходные данные в значимые результаты, «обучаясь» на известных примерах входных данных и результатов. То есть главной задачей машинного и глубокого обучения является *значимое преобразование данных*, или, иными словами, обучение *представлению* входных данных, приближающему нас к ожидаемому результату. Прежде чем двинуться дальше, давайте определим, что есть представление данных? По сути, это другой способ *представления*, или *кодирования*, данных. Например, цветное изображение можно закодировать в формате RGB (red-green-blue — красный-зеленый-синий) или HSV (hue-saturation-value — тон-насыщенность-значение): это два разных представления одних и тех же данных. Некоторые задачи трудно решаются с данными в одном представлении, но легко — в другом. Например, задача «выбрать все красные пикселя в изображении» легче решается с данными в формате RGB, тогда как задача «сделать изображение менее насыщенным» проще решается с данными в формате HSV. Главная задача моделей машинного обучения как раз и заключается в поиске соответствующего представления входных данных — преобразований, которые сделают данные более пригодными для решения такой задачи, как классификация.

Обратимся к конкретному примеру. Рассмотрим систему координат с осями X и Y и несколько точек в этой системе координат (x, y), как показано на рис. 1.3.

Как видите, у нас имеется несколько белых и черных точек. Допустим нам нужно разработать алгоритм, принимающий координаты (x, y) точки и возвращающий наиболее вероятный цвет, черный или белый. В данном случае:

- исходными данными являются координаты точек;
- результатом является цвет точек;
- мерой качества алгоритма может быть, например, процент правильно классифицированных точек.

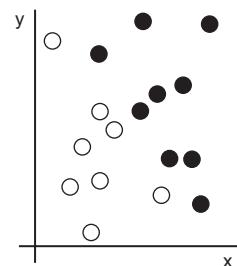


Рис. 1.3. Пример некоторых данных

В данном случае нам нужно получить новый способ представления исходных данных, позволяющий четко отделять белые точки от черных. Таким преобразованием, кроме прочих других, могло бы быть изменение системы координат, как показано на рис. 1.4.



Рис. 1.4. Изменение системы координат

Координаты наших точек в этой новой системе координат можно назвать новым представлением данных. Причем более удачным! Задачу классификации данных «черный/белый» в этом представлении можно свести к простому правилу: «черные точки имеют координату  $x > 0$ » или «белые точки имеют координату  $x < 0$ ». Это новое представление фактически решает поставленную задачу.

В данном примере мы определили изменение координат вручную. Но если бы вместо этого мы реализовали систематический поиск разных вариантов изменения системы координат с использованием процента правильно классифицированных точек в качестве обратной связи, мы получили бы версию машинного обучения. *Обучение* в контексте машинного обучения описывает автоматический процесс поиска лучшего представления.

Все алгоритмы машинного обучения заключаются в автоматическом поиске таких преобразований, которые смогут привести данные к виду, более пригодному для данной задачи. Эти операции могут изменять координаты, как мы только что видели, или выполнять линейные проекции (что может приводить к уничтожению информации), трансляцию, нелинейные операции (такие, как «выбрать все точки с координатой  $x > 0$ ») и т. д. Алгоритмы машинного обучения обычно не отличаются чем-то особенным; они просто выполняют поиск в предопределенном наборе операций, который называют *пространством гипотез*.

То есть технически машинное обучение — это поиск значимого представления некоторых входных данных в предопределенном пространстве возможностей с использованием сигнала обратной связи. Эта простая идея позволяет решать чрезвычайно широкий круг интеллектуальных задач: от распознавания речи до автоматического вождения автомобиля.

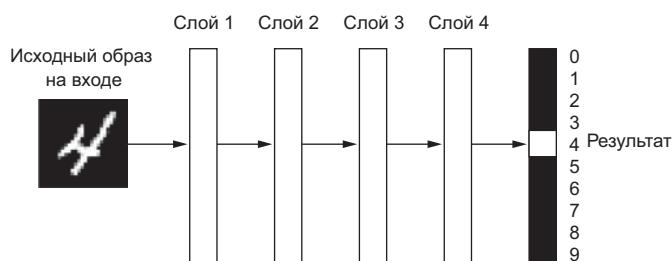
Теперь, когда вы получили представление о том, что понимается под *обучением*, давайте посмотрим, что особенного в *глубоком обучении*.

#### 1.1.4. «Глубина» глубокого обучения

Глубокое обучение — это особый раздел машинного обучения: новый подход к поиску представления данных, делающий упор на изучение последовательных слоев (или *уровней*) все более значимых представлений. Под *глубиной в глубоком обучении* не подразумевается более глубокое понимание, достигаемое этим подходом; идея заключается в многослойном представлении. Количество слоев, на которые делится модель данных, называют *глубиной* модели. Другими подходящими названиями для этой области машинного обучения могли бы служить: *многослойное обучение* и *иерархическое обучение*. Современное глубокое обучение часто вовлекает в процесс десятки и даже сотни последовательных слоев представления — и все они автоматически определяются под воздействием обучающих данных. Между тем другие подходы к машинному обучению ориентированы на изучении одного-двух слоев представления данных; по этой причине их иногда называют *поверхностным обучением*.

В глубоком обучении такие многослойные представления изучаются (почти всегда) с использованием моделей, так называемых *нейронных сетей*, структурированных в виде слоев, наложенных друг на друга. Термин *нейронная сеть* заимствован из нейробиологии, тем не менее, хотя некоторые основополагающие идеи глубокого обучения отчасти заимствованы из науки о мозге, модели глубокого обучения не являются моделями мозга. Нет никаких доказательств, что мозг реализует механизмы, подобные механизмам, используемым в современных моделях глубокого обучения. Вам могут встретиться научно-популярные статьи, в которых утверждается, что глубокое обучение работает подобно мозгу или моделирует работу мозга, но в действительности это не так. Было бы неправильно и контрпродуктивно заставлять начинающих освоение этой области думать, что глубокое обучение каким-то образом связано с нейробиологией; вам не нужно представление «как наш мозг», и вы также можете забыть все, что читали о гипотетической связи между глубоким обучением и биологией. Намного продуктивнее считать глубокое обучение математической основой для изучения представлений данных.

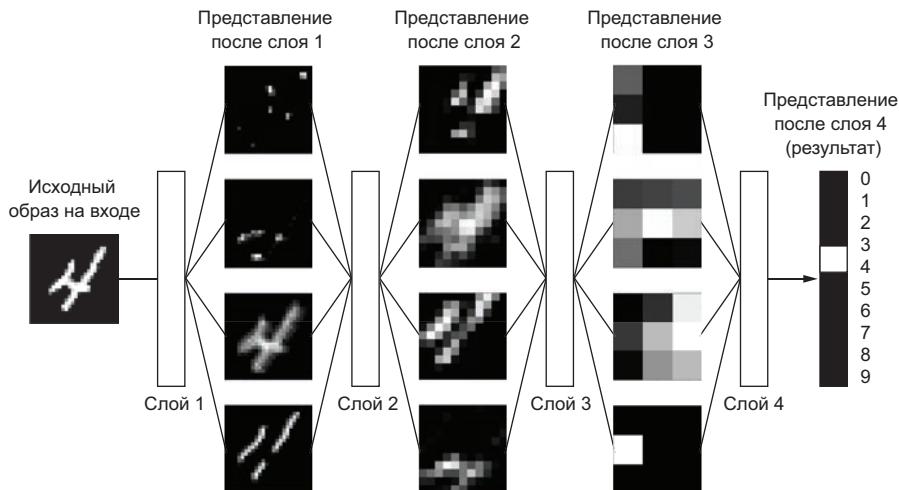
Как выглядят представления, получаемые алгоритмом глубокого обучения? Давайте исследуем, как сеть, имеющая несколько слоев в глубину (рис. 1.5), преобразует изображение цифры для ее распознавания.



**Рис. 1.5.** Глубокая нейронная сеть для классификации цифр

Как показано на рис. 1.6, сеть поэтапно преобразует образ цифры в представление, все больше отличающееся от исходного образа и несущее все больше полезной информации о результате. Глубокую сеть можно рассматривать как многоэтапную операцию очистки информации, в ходе которой информация проходит через последовательность фильтров и выходит из нее в *очищенном* виде (то есть в виде, пригодном для решения некоторых задач).

С технической точки зрения глубокое обучение — это многоступенчатый способ получения представления данных. Идея проста, но, как оказывается, очень простые механизмы при определенном масштабе могут выглядеть непонятными и таинственными.



**Рис. 1.6.** Глубокие представления, получаемые моделью классификации цифр

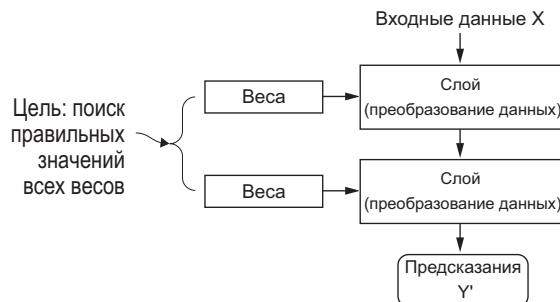
### 1.1.5. Принцип действия глубокого обучения в трех картинках

Теперь вы знаете, что суть машинного обучения заключается в преобразовании ввода (например, изображений) в результат (такой, как подпись «кошка»), которое выявляется путем исследования множества примеров входных данных и результатов. Вы также знаете, что глубокие нейронные сети превращают исходные данные в результат, выполняя длинную последовательность простых преобразований (слоев), и обучаются этим преобразованиям на примерах. Теперь посмотрим, как именно происходит обучение.

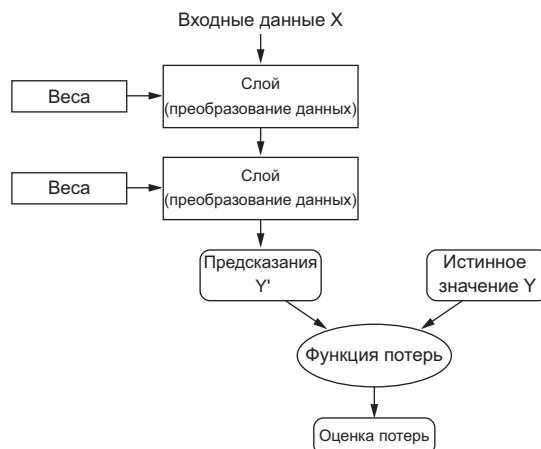
Что именно уровень делает со своими входными данными, определяется его *весами*, которые фактически являются набором чисел. Выражаясь техническим языком, можно сказать, что преобразование, реализуемое слоем, *параметризуется* его весами (рис. 1.7). (Веса также иногда называют *параметрами* слоя.) В данном контексте под *обучением* подразумевается поиск набора значений весов всех слоев в сети, при котором сеть будет правильно отображать образцовые входные данные в соответствующие им результаты. Но вот в чем дело: глубокая нейронная сеть может содержать десятки миллионов параметров. Поиск правильного значения для каждого из них может оказаться самой сложной задачей, особенно если изменение значения одного параметра влияет на поведение всех остальных!

Чтобы чем-то управлять, сначала нужно получить возможность наблюдать за этим. Чтобы управлять результатом работы нейронной сети, нужно иметь возможность измерить, насколько этот результат далек от ожидаемого. Эту задачу решает

*функция потерь* сети, которую также называют *целевой функцией*. Функция потерь принимает предсказание, выданное сетью, и истинное значение (которое сеть должна была вернуть) и вычисляет оценку расстояния между ними, отражающую, насколько хорошо сеть справилась с данным конкретным примером (рис. 1.8).



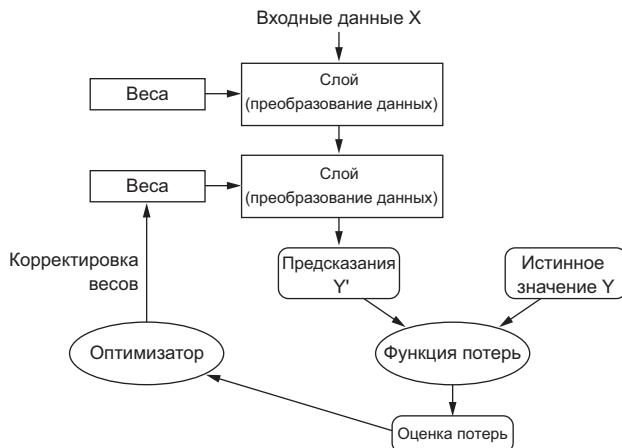
**Рис. 1.7.** Нейронная сеть параметризуется ее весами



**Рис. 1.8.** Функция потерь оценивает качество результатов, производимых нейронной сетью

Основная хитрость глубокого обучения заключается в использовании этой оценки для корректировки значений весов с целью уменьшения потерь в текущем примере (рис. 1.9). Данная корректировка является задачей *оптимизатора*, который реализует так называемый *алгоритм обратного распространения ошибки*: центральный алгоритм глубокого обучения. Подробнее об алгоритме обратного распространения ошибки рассказывается в следующей главе.

Первоначально весам сети присваиваются случайные значения, то есть фактически сеть реализует последовательность случайных преобразований. Естественно, получаемый ею результат далек от идеала, и оценка потерь, соответственно, очень



**Рис. 1.9.** Оценка потерь используется как обратная связь для корректировки весов

высока. Но с каждым примером, обрабатываемым сетью, веса корректируются в нужном направлении, и оценка потерь уменьшается. Это *цикл обучения*, который повторяется достаточное количество раз (обычно десятки итераций с тысячами примеров) и порождает весовые значения, минимизирующие функцию потерь. Сеть с минимальными потерями, возвращающая результаты, близкие к истинным, называется обученной сетью. Повторюсь еще раз: это простой механизм, который при определенном масштабе начинает выглядеть непонятным и таинственным.

### 1.1.6. Какой ступени развития достигло глубокое обучение

Несмотря на то что глубокое обучение является давним разделом машинного обучения, фактическое его развитие началось только в начале 2010-х. За прошедшие несколько лет в этой области произошла ни много ни мало революция, с особенно заметными успехами в моделировании восприятия — зрения и слуха — задач, кажущихся естественными и понятными для человека, но долгое время не дававшихся компьютерам.

В частности, глубокое обучение достигло следующих прорывов в традиционно сложных областях машинного обучения:

- ❑ классификация изображений на уровне человека;
- ❑ распознавание речи на уровне человека;
- ❑ распознавание рукописного текста на уровне человека;
- ❑ улучшение качества машинного перевода с одного языка на другой;
- ❑ улучшение качества машинного чтения текста вслух;

- ❑ появление цифровых помощников, таких как Google Now и Amazon Alexa;
- ❑ управление автомобилем на уровне человека;
- ❑ повышение точности целевой рекламы, используемой компаниями Google, Baidu и Bing;
- ❑ повышение релевантности поиска в интернете;
- ❑ появление возможности отвечать на вопросы, заданные вслух;
- ❑ игра в Го сильнее человека.

Мы все еще продолжаем исследовать возможности, которые таит в себе глубокое обучение. Мы начали применять его к широкому кругу проблем за пределами машинного восприятия и понимания естественного языка, таких как формальные рассуждения. Успех в этом направлении может означать начало новой эры, когда глубокое обучение будет помогать людям в науке, разработке программного обеспечения и многих других областях.

### 1.1.7. Не верьте рекламе

В сфере глубокого обучения за последние годы удалось добиться заметных успехов, однако ожидания на будущее десятилетие обычно намного превышают вероятные достижения. Даже при том, что многие значительные применения, такие как автопилоты для автомобилей, находятся практически на заключительной стадии реализации, многие другие, такие как полноценные диалоговые системы, перевод между произвольными языками на уровне человека и понимание естественного языка на уровне человека, скорее всего, еще долгое время будут оставаться недостижимыми. В частности, не стоит всерьез воспринимать разговоры об *интеллекте на уровне человека*. Завышенные ожидания от ближайшего будущего таят опасность: из-за невозможности реализации новых технологий инвестиции в исследования будут падать и прогресс замедлится на долгое время.

Такое уже происходило раньше. В прошлом ИИ пережил две волны подъема оптимизма, за которыми следовал спад, сопровождаемый разочарованиями и скептицизмом и, как результат, снижением финансирования. Все началось с символического ИИ в 1960-х. В те ранние годы давались весьма многообещающие прогнозы развития ИИ. Один из самых известных пионеров и сторонников подхода символического ИИ — Марвин Мински (Marvin Minsky) — в 1967 году заявил: «В течение поколения... проблема создания “искусственного интеллекта” будет практически решена». Три года спустя, в 1970-м, он сделал более точное предсказание: «Через три–восемь лет у нас появится машина с интеллектом среднего человека». В 2016 году это достижение все еще кажется далеким будущим — пока мы не можем предсказать, сколько времени уйдет на это, — но в 1960-х и в начале 1970-х некоторые эксперты (как и многие люди ныне) полагали, что это находится прямо за углом. Несколько лет спустя, из-за неоправдавшихся высоких ожиданий, исследователи и правительственные фонды отвернулись от этой области, — так

началась первая зима ИИ (вполне уместная метафора, потому что все это происходило вскоре после начала холодной войны).

Этот спад был не последним. В 1980-х начался подъем интереса к символическому ИИ благодаря буму экспертных систем в крупных компаниях. Первые успехи вызвали волну инвестиций, и корпорации по всему миру стали создавать свои отделы ИИ, занимающиеся разработкой экспертных систем. К 1985 году компании тратили более миллиарда долларов США в год на развитие технологии; но к началу 1990-х, из-за дороговизны в обслуживании, сложностей в масштабировании и ограниченности применения, интерес снова начал падать. Так началась вторая зима ИИ.

В настоящее время мы подходим к третьему циклу разочарований в ИИ, но пока мы еще находимся в фазе завышенного оптимизма. Сейчас лучше всего умерить наши ожидания на ближайшую перспективу и постараться донести до людей, мало знакомых с технической стороной этой области, что именно может дать глубокое обучение и на что оно не способно.

### 1.1.8. Перспективы ИИ

Даже при том, что наши ожидания на ближайшую перспективу могут быть нереалистичными, долгосрочная картина выглядит весьма ярко. Мы только начинаем применять глубокое обучение к решению многих важных задач: от медицинских диагнозов до цифровых помощников. В последние пять лет исследования в области ИИ продвигались удивительно быстро, во многом благодаря высокому уровню финансирования, никогда прежде не наблюдавшемуся в короткой истории ИИ, но пока слишком малому, чтобы этот прогресс воплотить в продукты и процессы, формирующие наш мир. Большинство результатов исследований в глубоком обучении пока не нашли практического применения, по крайней мере применения к решению полного спектра задач, где эта технология могла бы найти применение. Ваш доктор и ваш бухгалтер пока не используют ИИ. Вы сами в своей повседневной жизни тоже, вероятно, не используете технологии ИИ. Конечно, вы можете задавать простые вопросы своему смартфону и получать разумные ответы, вы можете получить весьма полезные рекомендации при выборе товаров на Amazon.com и по фразе «день рождения» быстро найти в Google Photos фотографии со дня рождения вашей дочери, который был в прошлом месяце. Это, несомненно, большой шаг вперед. Но такие инструменты лишь дополняют нашу жизнь. ИИ еще не занял центральное место в нашей жизни, работе и мыслях.

Сейчас трудно поверить, что ИИ может оказать значительное влияние на наш мир, потому что еще не развернулся во всю ширь, — так же как в 1995 году трудно было поверить в будущее влияние интернета. В то время большинство не понимало, какое отношение к ним может иметь интернет и как он изменит их жизнь. То же можно сегодня сказать о глубоком обучении и об искусственном интеллекте. Будьте уверены: эра ИИ наступит. В недалеком будущем ИИ станет вашим помощником и даже другом; он ответит на ваши вопросы, поможет воспитывать детей и про-

следит за здоровьем. Он доставит продукты к вашей двери и отвезет вас из пункта А в пункт Б. Это будет ваш интерфейс к миру, который все более усложняется и наполняется информацией. И, что особенно важно, ИИ будет способствовать человечеству в движении вперед, помогая ученым делать новые прорывные открытия во всех областях науки, от геномики до математики.

По пути мы можем столкнуться с неудачами и, возможно, пережить новую зиму ИИ — так же как после всплеска развития интернет-индустрии в 1998–1999 годах произошел спад, вызванный уменьшением инвестиций в начале 2000-х. Но мы приедем туда рано или поздно. В конечном итоге ИИ будет применяться во всех процессах в нашем обществе и в нашей жизни, так же как интернет сегодня.

Не верьте рекламе, но доверяйте долгосрочным прогнозам. Может потребоваться какое-то время, пока ИИ раскроет весь свой потенциал, глубину которого пока еще никто не может даже представить, но ИИ придет и изменит наш мир фантастическим образом.

## 1.2. Что было до глубокого обучения: краткая история машинного обучения

Глубокое обучение достигло уровня общественного внимания и инвестиций, не виданных прежде в истории искусственного интеллекта, но это не первая успешная форма машинного обучения. Можно с уверенностью сказать, что большинство алгоритмов машинного обучения, используемых сейчас в промышленности, не являются алгоритмами глубокого обучения. Глубокое обучение не всегда является правильным инструментом — иногда просто недостаточно данных для глубокого обучения, иногда проблема лучше решается с применением других алгоритмов. Если глубокое обучение — ваш первый контакт с машинным обучением, вы можете оказаться в ситуации, когда, получив в руки молоток глубокого обучения, вы начнете воспринимать все задачи машинного обучения как гвозди. Единственный способ не попасть в эту ловушку — познакомиться с другими подходами и практиковать их, когда это необходимо.

В этой книге мы не будем подробно обсуждать классические подходы к машинному обучению, но коротко представим их и опишем исторический контекст, в котором они разрабатывались. Это поможет вам увидеть, какое место занимает глубокое обучение в более широком контексте машинного обучения, и лучше понять, откуда пришло глубокое обучение и почему оно имеет большое значение.

### 1.2.1. Вероятностное моделирование

*Вероятностное моделирование* — это применение принципов статистики к анализу данных. Это одна из самых ранних форм машинного обучения, которая до сих пор находит широкое использование. Одним из наиболее известных алгоритмов в этой категории является наивный байесовский алгоритм.

Наивный байесовский алгоритм — это вид классификатора машинного обучения, основанный на применении теоремы Байеса со строгими (или «наивными», откуда и происходит название алгоритма) предположениями о независимости входных данных. Эта форма анализа данных предшествовала появлению компьютеров и десятилетиями применялась вручную, пока не появилась ее первая реализация на компьютере (в 1950-х годах). Теорема Байеса и основы статистики были заложены в XVIII столетии, и это все, что нужно для использования наивных байесовских классификаторов.

С этим алгоритмом тесно связана модель *логистической регрессии* (сокращенно *logreg*), которую иногда рассматривают как аналог примера «hello world» в машинном обучении. Пусть вас не вводят в заблуждение название. Модель логистической регрессии — это алгоритм классификации, а не регрессии. Так же как наивный байесовский алгоритм, модель логистической регрессии была разработана задолго до появления компьютеров, но до сих пор остается востребованной благодаря своей простоте и универсальной природе. Часто это первое, что пытается сделать исследователь со своим набором данных, чтобы получить представление о классификации.

## 1.2.2. Первые нейронные сети

Ранние версии нейронных сетей были полностью вытеснены современными вариантами, о которых рассказывается на страницах этой книги, но вам будет полезно знать, как возникло глубокое обучение. Основные идеи нейронных сетей в упрощенном виде были исследованы еще в 1950-х. Долгое время развитие этого подхода тормозилось из-за отсутствия эффективного способа обучения больших нейронных сетей. Но ситуация изменилась в середине 1980-х, когда несколько исследователей, независимо друг от друга, вновь открыли алгоритм обратного распространения ошибки — способ обучения цепочек параметрических операций с использованием метода градиентного спуска (далее в книге мы дадим точные определения этим понятиям) — и начали применять его к нейронным сетям.

Первое успешное практическое применение нейронных сетей датируется 1989 годом, когда Ян Лекун (Yann LeCun) в Bell Labs объединил ранние идеи сверточных нейронных сетей и обратного распространения ошибки и применил их для решения задачи распознавания рукописных цифр. Получившаяся в результате нейронная сеть была названа *LeNet* и использовалась почтовой службой США в 1990-х для автоматического распознавания почтовых индексов на конвертах.

## 1.2.3. Ядерные методы

По мере привлечения внимания исследователей к нейронным сетям в 1990-х и благодаря первому успеху приобрел известность новый подход к машинному обучению, быстро отправивший нейронные сети обратно в небытие: ядерные методы (*kernel methods*). *Ядерные методы* — это группа алгоритмов классификации, из

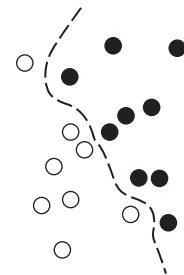
которых наибольшую известность получил *метод опорных векторов* (Support Vector Machine, SVM). Современная формулировка SVM была предложена Владимиром Вапником (Vladimir Vapnik) и Коринной Кортес (Corinna Cortes) в начале 1990-х в Bell Labs и опубликована в 1995 году<sup>1</sup>. Прежняя линейная формулировка была опубликована Вапником и Алексеем Червоненкисом (Alexeey Chervonenkis) в начале 1963 года<sup>2</sup>.

Метод опорных векторов предназначен для решения задач классификации путем поиска хороших *решающих границ* (рис. 1.10), разделяющих два набора точек, принадлежащих разным категориям. Решающей границей может быть линия или поверхность, разделяющая выборку обучающих данных на пространства, принадлежащие двум категориям. Для классификации новых точек достаточно только проверить, по какую сторону от границы они находятся.

Поиск таких границ метод опорных векторов осуществляет в два этапа:

1. Данные отображаются в новое пространство более высокой размерности, где граница может быть представлена как гиперплоскость (если данные были двумерными, как на рис. 1.10, гиперплоскость вырождается в линию).
2. Хорошая решающая граница (разделяющая гиперплоскость) вычисляется путем максимизации расстояния от гиперплоскости до ближайших точек каждого класса, этот этап называют *максимизацией зазора*. Это позволяет обобщить классификацию новых образцов, не принадлежащих обучающему набору данных.

Методика отображения данных в пространство более высокой размерности, где задача классификации становится проще, может хорошо выглядеть на бумаге, но на практике часто оказывается трудноприменимой. Вот тут и приходит на помощь изящная процедура *kernel trick* (ключевая идея, по которой ядерные методы получили свое название). Суть ее заключается в следующем: чтобы найти хорошие решающие гиперплоскости в новом пространстве, явно вычислять координаты точек в этом пространстве не требуется; достаточно вычислить расстояния между парами точек, что можно эффективно сделать с помощью *функции ядра*. Функция ядра — это незатратная вычислительная операция, которая отображает любые две точки из исходного пространства и вычисляет расстояние между ними в целевом пространстве представления, полностью минуя явное вычисление нового представления. Функции ядра обычно определяются вручную, а не извлекаются из



**Рис. 1.10.**  
Решающая  
граница

<sup>1</sup> Vladimir Vapnik and Corinna Cortes, «Support-Vector Networks», Machine Learning 20, no. 3 (1995): 273–297.

<sup>2</sup> Vladimir Vapnik and Alexey Chervonenkis, «A Note on One Class of Perceptrons», Automation and Remote Control 25 (1964).

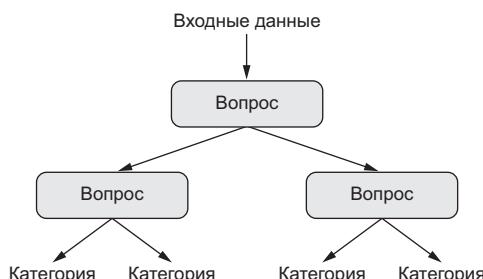
данных — в случае с методом опорных векторов по данным определяется только разделяющая гиперплоскость.

На момент разработки метод опорных векторов демонстрировал лучшую производительность на простых задачах классификации и был одним из немногих методов машинного обучения, обладающих обширной теоретической базой и поддающихся серьезному математическому анализу, что сделало его хорошо понятным и легко интерпретируемым. Благодаря этим свойствам метод опорных векторов приобрел чрезвычайную популярность на долгие годы.

Однако метод опорных векторов оказался трудно применимым к большим наборам данных и не дал хороших результатов для таких задач, как классификация изображений. Так как SVM является поверхностным методом, для его применения к задачам распознавания требуется сначала вручную выделить представительную выборку (этот шаг называется *конструированием признаков*), что сопряжено со сложностями и чревато ошибками.

#### 1.2.4. Деревья решений, случайные леса и градиентный бустинг

*Деревья решений* — это иерархические структуры, которые позволяют классифицировать входные данные или предсказывать выходные значения по заданным исходным значениям (рис. 1.11). Они легко визуализируются и интерпретируются. Деревья решений, формируемые на основе данных, заинтересовали исследователей в 2000-х, и к 2010-м им часто отдавали предпочтение перед ядерными методами.



**Рис. 1.11.** Дерево решений: обучаемыми параметрами являются вопросы о данных. Таким вопросом мог бы быть, например, вопрос «Коэффициент 2 в данных больше 3,5?»

В частности, алгоритм *случайный лес* (Random Forest) предложил надежный и практичный подход к обучению на основе деревьев решений, включающий в себя создание большого количества специализированных деревьев решений с последующим объединением выдаваемых ими результатов. Случайные леса применимы

к широкому кругу задач — можно сказать, что они почти всегда являются оптимальным алгоритмом для любых задач поверхностного машинного обучения. Когда в 2010 году был запущен популярный конкурсный веб-сайт Kaggle (<http://kaggle.com>), посвященный машинному обучению, случайные леса быстро превратились в наиболее популярную платформу, пока в 2014 году не появился *метод градиентного бустинга*. Метод градиентного бустинга, во многом напоминающий случайный лес, — это прием машинного обучения, основанный на объединении слабых моделей прогнозирования, обычно — деревьев решений. Он использует градиентный бустинг, способ улучшения любой модели машинного обучения путем итеративного обучения новых моделей, специализированных для устранения слабых мест в предыдущих моделях. Применительно к деревьям решений прием градиентного бустинга позволяет получить модели, которые в большинстве случаев превосходят случайные леса при сохранении аналогичных свойств. На сегодняшний день это один из лучших алгоритмов, хотя и *не самый* лучший, для решения задач, не связанных с распознаванием. Наряду с глубоким обучением, это один из наиболее широко используемых приемов на сайте Kaggle.

### 1.2.5. Назад к нейронным сетям

Примерно в 2010 году, несмотря на почти полную потерю интереса к нейронным сетям со стороны научного сообщества, ряд исследователей, продолжавших работать с нейронными сетями, стали добиваться важных успехов: группы Джейфри Хинтона (Geoffrey Hinton) из университета в Торонто, Йошуа Бенгио (Yoshua Bengio) из университета в Монреале, Яна Лекуна (Yann LeCun) из университета в Нью-Йорке и исследователи в научно-исследовательском институте искусственного интеллекта IDSIA в Швейцарии.

В 2011 году Ден Киресан (Dan Ciresan) из IDSIA выиграл академический конкурс по классификации изображений с применением глубоких нейронных сетей, обучаемых на GPU, — это был первый практический успех современного глубокого обучения. Но перелом произошел в 2012 году, когда группа Хинтона приняла участие в ежегодном соревновании ImageNet по крупномасштабному распознаванию образов. ImageNet предложила очень сложное на то время задание, заключающееся в классификации цветных изображений с высоким разрешением на 1000 разных категорий после обучения по выборке, включающей в себя 1,4 миллиона изображений. В 2011 году модель-победитель, основанная на классических подходах к распознаванию образов, показала точность лишь 74,3 %. В 2012 году команда Алекса Крижевски (Alex Krizhevsky), советником в которой был Джейфри Хинтон (Geoffrey Hinton), достигла точности в 83,6 % — значительный прорыв. С тех пор каждый год первые позиции в этом соревновании занимают глубокие сверточные нейронные сети. В 2015 году победитель достиг точности в 96,4 %, и задача классификации на ImageNet была сочтена решенной полностью.

Начиная с 2012 года глубокие сверточные нейронные сети (сокращенно *convnets*) перешли в разряд передовых алгоритмов для всех задач распознавания образов;

в более общем плане они с успехом могут использоваться в любых задачах распознавания. На крупных конференциях по распознаванию образов, проводившихся в 2015 и 2016 годах, было трудно найти презентацию, не включающую сверточных нейросетей в том или ином виде. В то же время глубокое обучение нашло применение во многих других видах задач, таких как обработка естественного языка. Оно полностью заменило метод опорных векторов и деревья решений в широком круге задач. Например, в течение нескольких лет Европейская организация по ядерным исследованиям (European Organization for Nuclear Research, CERN) использовала методы на основе деревьев решений для данных, получаемых с детектора частиц ATLAS в большом адронном коллайдере; но затем в CERN было принято решение перейти на использование глубоких нейронных сетей на основе Keras из-за их более высокой производительности и простоты обучения на больших наборах данных.

### 1.2.6. Отличительные черты глубокого обучения

Основная причина быстрого взлета глубокого обучения заключается в лучшей производительности во многих задачах. Однако это не единственная причина. Глубокое обучение также существенно упрощает решение проблем, полностью автоматизируя важнейший шаг в машинном обучении, выполнявшийся раньше вручную, — *конструирование признаков*.

Предшествовавшие методы машинного обучения — методы поверхностного обучения — включали в себя преобразование входных данных только в одно или два последовательных пространства, обычно посредством простых преобразований, таких как нелинейная проекция в пространство более высокой размерности (метод опорных векторов) или деревья решений. Однако точные представления, необходимые для решения сложных задач, обычно нельзя получить такими способами. Поэтому людям приходилось прилагать большие усилия, чтобы привести исходные данные к виду, более пригодному для обработки этими методами: им приходилось вручную улучшать слой представления своих данных. Это называется *конструированием признаков*. Глубокое обучение, напротив, полностью автоматизирует этот шаг: с применением методов глубокого обучения все признаки извлекаются за один проход, без необходимости конструировать их вручную. Это очень упростило процесс машинного обучения, потому что часто сложный и многоступенчатый конвейер оказалось возможным заменить единственной простой сквозной моделью глубокого обучения.

Вы можете спросить: если суть вопроса заключается в получении нескольких последовательных слоев представлений, можно ли многократно применить методы поверхностного обучения для имитации эффекта глубокого обучения? На практике наблюдается быстрое уменьшение последовательных применений методов поверхностного обучения, поскольку *оптимальный слой первого представления в трехслойной модели не является оптимальным первым слоем в однослоиной или двухслойной модели*. Особенность преобразования в глубоком обучении состоит в том, что модель может исследовать все слои представления *вместе* и одновре-

менно, а не последовательно (последовательное исследование также называют «жадным»). При совместном изучении признаков, когда модель корректирует один из своих внутренних признаков, все прочие признаки, зависящие от него, автоматически корректируются в соответствии с изменениями, без вмешательства человека. Все контролируется единственным сигналом обратной связи: каждое изменение в модели служит конечной цели. Это намного более мощный подход, чем жадно накладывать поверхностные модели друг на друга, потому что он позволяет изучать более сложные абстрактные представления, разбивая их на длинные ряды промежуточных пространств (слоев), в которых каждое последующее пространство получается в результате простого преобразования предыдущего.

Методика глубокого обучения обладает двумя важными характеристиками: она *поэтапно, послойно конструирует все более сложные представления и совместно исследует промежуточные представления*, благодаря чему каждый слой обновляется в соответствии с потребностями представления слоя выше и потребностями слоя ниже. Вместе эти два свойства делают глубокое обучение намного успешнее предыдущих подходов к машинному обучению.

### 1.2.7. Современный ландшафт машинного обучения

Конкурсный сайт Kaggle дает отличную возможность получить представление о текущем ландшафте алгоритмов и инструментов машинного обучения. Благодаря соревновательному характеру (в некоторых конкурсах участвуют тысячи соискателей, а призы составляют миллионы долларов США) и широкому разнообразию задач машинного обучения, на сайте Kaggle можно реально оценить, какие подходы используются и насколько успешно. Так какой же алгоритм уверенно выигрывает состязания? Какими инструментами пользуются победители?

В 2016 и 2017 годах на сайте Kaggle главенствовали два подхода: метод градиентного бустинга и глубокое обучение. Метод градиентного бустинга, в частности, использовался для решения задач, в которых присутствовали структурированные данные, тогда как глубокое обучение использовалось для решения задач распознавания, таких как классификация изображений. Те, кто практикует первый подход, почти всегда используют великолепную библиотеку XGBoost, предлагающую поддержку двух языков, наиболее популярных в науке о данных: Python и R. Между тем большинство конкурсантов, практикующих глубокое обучение, используют библиотеку Keras — простую в применении, гибкую и поддерживающую язык Python.

Этим двум методам вы должны уделять особое внимание, чтобы добиться успеха в применении машинного обучения в наше время: метод градиентного бустинга для задач поверхностного обучения и глубокое обучение для задач распознавания. В техническом плане это означает, что вы должны владеть двумя библиотеками — XGBoost и Keras, занимающими доминирующее положение в конкурсах на сайте Kaggle. Как только вы взяли в руки эту книгу, вы уже сделали большой шаг к этой цели.

## 1.3. Почему глубокое обучение? Почему сейчас?

Две ключевые идеи глубокого обучения для решения задач распознавания образов — сверточные нейронные сети и алгоритм обратного распространения ошибки — были хорошо известны уже в 1989 году. Алгоритм долгой краткосрочной памяти (Long Short-Term Memory, LSTM), составляющий основу глубокого обучения для прогнозирования временных рядов, был предложен в 1997 году и с тех пор почти не изменился. Так почему же глубокое обучение начало применяться только с 2012 года? Что изменилось за эти два десятилетия?

В целом машинным обучением движут три технические силы:

- оборудование;
- наборы данных и тесты;
- алгоритмические достижения.

Поскольку эта область руководствуется экспериментальными выводами, а не теорией, алгоритмические достижения возможны только при наличии данных и оборудования, пригодных для проверки идей (или, как это часто бывает, для возрождения старых идей). Машинное обучение — это не математика и не физика, где прорывы могут быть сделаны с помощью ручки и бумаги. Это инженерная наука.

Настоящим узким местом на протяжении 1990-х и 2000-х годов были данные и оборудование. Но в течение этого времени происходило бурное развитие интернета, а для рынка игрового программного обеспечения были созданы высокопроизводительные графические процессоры.

### 1.3.1. Оборудование

Между 1990 и 2010 годами быстродействие стандартных процессоров выросло примерно в 5000 раз. В результате сейчас на ноутбуке можно запускать небольшие модели глубокого обучения, тогда как 25 лет назад это в принципе было невозможно.

Но типичные модели глубокого обучения, используемые для распознавания образов или речи, требуют вычислительной мощности на порядки больше, чем мощность ноутбука. В течение 2000-х такие компании, как NVIDIA и AMD, вложили миллионы долларов в разработку быстрых процессоров с массовым параллелизмом (графических процессоров — Graphical Processing Unit [GPU]) для поддержки графики все более реалистичных видеоигр — недорогих специализированных суперкомпьютеров, предназначенных для отображения на экране сложных трехмерных сцен в режиме реального времени. Эти инвестиции принесли пользу научному сообществу, когда в 2007 году компания NVIDIA выпустила CUDA (<https://developer.nvidia.com/about-cuda>), программный интерфейс для линейки своих GPU. Несколько GPU теперь могут заменить мощные кластеры на обычных процессорах в различных задачах, допускающих возможность массового распараллеливания вычислений, начиная с физического моделирования. Глубокие нейронные сети,

выполняющие в основном умножение множества маленьких матриц, также допускают высокую степень распараллеливания; и ближе к 2011 году некоторые исследователи начали писать CUDA-реализации нейронных сетей. Одними из первых стали Дэн Кайесан (Dan Ciresan)<sup>1</sup> и Алекс Крижевски (Alex Krizhevsky)<sup>2</sup>.

Получилось так, что игровая индустрия субсидировала создание суперкомпьютеров для следующего поколения приложений искусственного интеллекта. Иногда крупные достижения начинаются с игр. Современный графический процессор NVIDIA TITAN X, стоивший 1000 долларов США в конце 2015 года, способен выдать пиковую производительность 6,6 терафлопса с одинарной точностью: 6,6 триллиона операций в секунду с числами типа `float32`. Это почти в 350 раз больше производительности современного ноутбука. Графическому процессору TITAN X требуется всего несколько дней для обучения модели ImageNet, выигравшей конкурс ILSVRC несколько лет тому назад. Между тем большие компании обучают модели глубокого обучения на кластерах, состоящих из сотен GPU, разработанных специально для нужд глубокого обучения, таких как NVIDIA Tesla K80. Эти кластеры обладают такой вычислительной мощностью, которая не была возможна без современных GPU.

Более того, индустрия глубокого обучения начинает выходить за рамки GPU и инвестировать средства в развитие еще более специализированных процессоров, наиболее эффективно показывающих себя в области глубокого обучения. В 2016 году на своей ежегодной конференции Google I/O компания Google продемонстрировала свой проект тензорного процессора (Tensor Processing Unit, TPU): процессор с новой архитектурой, предназначенный для использования в глубоких нейронных сетях, который, как сообщалось, в 10 раз производительнее и энергоэффективнее, чем топовые модели GPU.

### 1.3.2. Данные

Иногда ИИ объявляется новой индустриальной революцией. Если глубокое обучение — паровой двигатель этой революции, то данные — это уголь: сырье,итающее наши интеллектуальные машины, без которого невозможно движение вперед. К вопросу о данных: вдобавок к экспоненциальному росту емкости устройств хранения информации, наблюдавшемуся в последние 20 лет (согласно закону Мура), перемены в игровом мире вызвали бурный рост интернета, благодаря чему появилась возможность накапливать и распространять очень большие объемы данных для машинного обучения. В настоящее время крупные компании работают с коллекциями изображений, видео и текстовых материалов, которые невозможно было бы собрать без интернета. Например, изображения на сайте Flickr,

<sup>1</sup> См. статью «Flexible, High Performance Convolutional Neural Networks for Image Classification» в материалах 22-й международной конференции по искусственному интеллекту (2011), [www.ijcai.org/Proceedings/11/Papers/210.pdf](http://www.ijcai.org/Proceedings/11/Papers/210.pdf).

<sup>2</sup> См. статью «ImageNet Classification with Deep Convolutional Neural Networks» в журнале «Advances in Neural Information Processing Systems», № 25 (2012), <http://mng.bz/2286>.

классифицированные пользователями, стали золотой жилой для разработчиков моделей распознавания образов. То же можно сказать о видеороликах на YouTube. А Википедия стала ключевым источником наборов данных для задач обработки естественного языка.

Если и есть набор данных, ставший катализатором для развития глубокого обучения, то это коллекция ImageNet, включающая в себя 1,4 миллиона изображений, классифицированных вручную на 1000 категорий (каждое изображение отнесено только к одной категории). Но особенной коллекции ImageNet делает не только ее огромный размер, но также ежегодные соревнования<sup>1</sup>, в которых она задействована.

Как показывает пример Kaggle, публичные конкурсы — отличный способ мотивации исследователей и инженеров к преодолению все новых и новых рубежей. Наличие общих критериев оценки достижений исследователей значительно помогло недавнему росту глубокого обучения.

### 1.3.3. Алгоритмы

Кроме оборудования и данных, до конца 2000-х нам не хватало надежного способа обучения очень глубоких нейронных сетей. Как результат, нейронные сети оставались очень неглубокими, имеющими один или два слоя представления; в связи с этим они не могли противостоять более совершенным поверхностным методам, таким как метод опорных векторов и случайные леса. Основная проблема заключалась в *распространении градиента* через глубокие пакеты слоев. Сигнал обратной связи, используемый для обучения нейронных сетей, затухает по мере увеличения количества слоев.

Ситуация изменилась в 2009–2010 годах с появлением некоторых простых, но важных алгоритмических усовершенствований, позволивших улучшить распространение градиента:

- ❑ улучшенные функции активации;
- ❑ улучшенные схемы инициализации весов, начиная с предварительного послойного обучения, от которого быстро отказались;
- ❑ улучшенные схемы оптимизации, такие как RMSProp и Adam.

Только когда эти улучшения позволили создавать модели с 10 слоями и более, началось развитие глубокого обучения.

Наконец, в 2014, 2015 и 2016 годах были открыты еще более совершенные способы распространения градиента, такие как пакетная нормализация, обходные связи и отделимые свертки. В настоящее время мы можем обучать с нуля модели с тысячами слоев в глубину.

---

<sup>1</sup> Соревнования по распознаванию изображений ImageNet Large Scale Visual Recognition Challenge (ILSVRC), [www.image-net.org/challenges/LSVRC](http://www.image-net.org/challenges/LSVRC).

### 1.3.4. Новая волна инвестиций

Как отметили ведущие исследователи, в 2012–2013 годах глубокое обучение вывело на новый современный слой распознавание образов и в конечном счете все задачи распознавания. За этим последовала постепенно нарастающая волна инвестиций в индустрию, намного превосходящая все предыдущие, наблюдавшиеся в истории ИИ.

В 2011 году, как раз перед тем, как глубокое обучение вышло на лидирующие позиции, общие инвестиции венчурного капитала в ИИ составили около 19 миллионов долларов, которые почти полностью пошли на практическое применение методов поверхностного машинного обучения. К 2014 году они выросли до ошеломляющих 394 миллионов. За эти три года появились десятки стартапов, пытающихся извлечь выгоду из поднявшейся шумихи. Между тем крупные компании, такие как Google, Facebook, Baidu и Microsoft, инвестировали деньги в исследования, проводившиеся внутренними подразделениями, и объемы этих инвестиций почти наверняка превысили инвестиции венчурного капитала. Вот лишь несколько цифр, лежащих на поверхности: в 2013-м компания Google приобрела стартап DeepMind, занимающийся глубоким обучением, за 500 миллионов — крупнейшая сумма за приобретение компании в истории ИИ. В 2014-м компания Baidu открыла в Кремниевой долине свой центр по исследованию глубокого обучения, инвестировав в проект 300 миллионов долларов. В 2016-м компания Intel приобрела стартап Nervana Systems, занимающийся разработкой оборудования для глубокого обучения, за более чем 400 миллионов долларов.

Машинное обучение — и глубокое обучение в частности — заняло центральное место среди стратегических продуктов этих технологических гигантов. В конце 2015 года генеральный директор Google Сундар Пичаи (Sundar Pichai) отметил: «Машинное обучение — это основа для решительной смены системы координат в оценивании всей нашей деятельности. Мы вдумчиво применяем его во всех наших продуктах, будь то поиск, реклама, YouTube или Play. И мы с самого начала — и систематически — применяем машинное обучение во всех этих областях»<sup>1</sup>.

В результате этой волны инвестиций за прошедшие пять лет количество людей, работающих над глубоким обучением, увеличилось с нескольких сотен до десятков тысяч, а прогресс в исследованиях достиг небывалого уровня. В настоящее время пока не наблюдается признаков замедления этой тенденции в ближайшее время.

### 1.3.5. Демократизация глубокого обучения

Одним из ключевых факторов, обусловивших приток новых лиц в глубокое обучение, стала демократизация инструментов, используемых в этой области. На начальном этапе глубокое обучение требовало значительных знаний и опыта программирования на C++ и владения CUDA, чем могли похвастаться очень немногие.

---

<sup>1</sup> Sundar Pichai, Alphabet earnings call, Oct. 22, 2015.

В настоящее время для исследований в области глубокого обучения достаточно базовых навыков программирования на Python. Это вызвано, прежде всего, развитием Theano, а затем TensorFlow — двух фреймворков для Python, реализующих операции с тензорами, которые поддерживают автоматическое дифференцирование и значительно упрощают реализацию новых моделей, и появлением дружественных библиотек, таких как Keras, которые делают работу с глубоким обучением таким же простым делом, как манипулирование кубиками LEGO. После выхода в 2015 году библиотека Keras быстро была принята за основу многими новыми стартапами, аспирантами и исследователями, работающими в этой области.

### 1.3.6. Ждать ли продолжения этой тенденции?

Есть ли что-то особенное в глубоком обучении, что делает его «правильным» выбором и для компаний, вкладывающих инвестиции, и для исследователей? Или глубокое обучение — это просто увлечение, которое не продлится долго? Будем ли мы использовать глубокие нейронные сети через 20 лет?

Глубокое обучение имеет несколько свойств, которые оправдывают его статус как революции в ИИ, и оно задержится надолго. Возможно, нейронные сети исчезнут через два десятилетия, но все, что останется взамен, будет прямым наследником современного глубокого обучения и его основных идей. Эти важнейшие свойства можно разделить на три категории:

- ❑ *Простота* — глубокое обучение избавляет от необходимости конструировать признаки, заменяя сложные, противоречивые и тяжелые конвейеры простыми обучаемыми моделями, которые обычно строятся с использованием пяти-шести тензорных операций.
- ❑ *Масштабируемость* — глубокое обучение легко поддается распараллеливанию на GPU или TPU, поэтому оно в полной мере может использовать закон Мура. Кроме того, обучение моделей можно производить итеративно, на небольших пакетах данных, что дает возможность проводить обучение на наборах данных произвольного размера. (Единственным узким местом является объем доступной вычислительной мощности для параллельных вычислений, которая, как следует из закона Мура, является быстро перемещающимся барьером.)
- ❑ *Гибкость и готовность к многократному использованию* — в отличие от многих предшествовавших подходов, модели глубокого обучения могут обучаться на дополнительных данных без полного перезапуска, что делает их пригодными для непрерывного и продолжительного обучения — очень важное свойство для очень больших промышленных моделей. Кроме того, обучаемые модели глубокого обучения можно перенацеливать и, соответственно использовать многократно: например, модель, обученную классификации изображений, можно включить в конвейер обработки видео. Это позволяет использовать предыдущие наработки для создания все более сложных и мощных моделей. Это также позволяет применять глубокое обучение к очень маленьким объемам данных.

Глубокое обучение находится в центре внимания всего несколько лет, и мы еще не определили границы его возможностей. Каждый месяц мы узнаем о новых и новых вариантах использования и инженерных усовершенствованиях, которые снимают предыдущие ограничения. После научной революции прогресс обычно развивается по сигмоиде: сначала наблюдается быстрый рост, который постепенно стабилизируется, когда исследователи сталкиваются с труднопреодолимыми ограничениями, и затем дальнейшие усовершенствования замедляются. В 2017 году глубокое обучение, по всей видимости, находилось на первой половине этой сигмоиды, а это значит, что в следующие несколько лет можно ожидать еще большего прогресса.

# 2

## Прежде чем начать: математические основы нейронных сетей

Эта глава охватывает следующие темы:

- ✓ первый пример нейронной сети;
- ✓ тензоры и операции с тензорами;
- ✓ процесс обучения нейронной сети методами обратного распространения ошибки и градиентного спуска.

Для понимания глубокого обучения необходимо знать множество простых математических понятий: тензоры, операции с тензорами, дифференцирование, градиентный спуск и т. д. Наша цель в этой главе — познакомиться с этими понятиями, не погружаясь слишком глубоко в теорию. В частности, мы будем избегать математических формул, которые не всегда нужны для достаточно полного объяснения и могут оттолкнуть читателей, не имеющих математической подготовки.

Чтобы вам проще было разобраться с тензорами и градиентным спуском, мы начнем главу с практического примера нейронной сети. А затем станем постепенно знакомиться с новыми понятиями. Имейте в виду, что знание этих понятий потребуется вам для понимания практических примеров в следующих главах!

Прочитав эту главу, вы будете понимать, как действуют нейронные сети, и сможете перейти к изучению приемов практического применения, которые начнутся с главы 3.

### 2.1. Первое знакомство с нейронной сетью

Рассмотрим конкретный пример нейронной сети, которая обучается классификации рукописных цифр и создана с помощью библиотеки Keras для Python. Если у вас нет опыта использования Keras или других подобных библиотек, возможно,

вы не все поймете в этом первом примере. Может быть, вы еще не установили Keras; в этом нет ничего страшного. В следующей главе мы рассмотрим каждый элемент в примере и подробно объясним их. Поэтому не волнуйтесь, если какие-то шаги покажутся непонятными или похожими на магию. В конце концов, мы должны с чего-то начать.

Перед нами стоит задача: реализовать классификацию черно-белых изображений рукописных цифр ( $28 \times 28$  пикселов) по 10 категориям (от 0 до 9). Мы будем использовать набор данных MNIST, популярный в сообществе исследователей глубокого обучения, который существует практически столько же, сколько сама область машинного обучения, и широко используется для обучения. Этот набор содержит 60 000 обучающих изображений и 10 000 контрольных изображений, собранных Национальным институтом стандартов и технологий США (National Institute of Standards and Technology — часть NIST в аббревиатуре MNIST) в 1980-х. «Решение» задачи MNIST можно рассматривать как своеобразный аналог «Hello World» в глубоком обучении — часто это первое действие, которое выполняется, чтобы убедиться, что алгоритмы действуют в точности как ожидалось. По мере углубления в практику машинного обучения вы увидите, что MNIST часто упоминается в научных статьях, блогах и т. д. Некоторые образцы изображений из набора MNIST можно видеть на рис. 2.1.

### ПРИМЕЧАНИЕ О КЛАССАХ И МЕТКАХ

В машинном обучении *категория* в задаче классификации называется *классом*. Элементы исходных данных называются *образцами*. Класс, связанный с конкретным образцом, называется *меткой*.



**Рис. 2.1.** Образцы изображений MNIST

Не пытайтесь сразу же воспроизвести этот пример на своем компьютере. Чтобы его опробовать, нужно сначала установить библиотеку Keras, о чём будет рассказано позже в разделе 3.3.

Набор данных MNIST уже входит в состав Keras в форме набора из четырех массивов Numpy.

### Листинг 2.1. Загрузка набора данных MNIST в Keras

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Здесь `train_images` и `train_labels` — это *тренировочный набор*, то есть данные, необходимые для обучения. После обучения модель будет проверяться тестовым (или контрольным) набором, `test_images` и `test_labels`.

Изображения хранятся в массивах NumPy, а метки — в массиве цифр от 0 до 9. Изображения и метки находятся в прямом соответствии, один к одному.

Рассмотрим обучающие данные:

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

И контрольные данные:

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Вот как мы будем действовать дальше: сначала передадим нейронной сети обучающие данные, `train_images` и `train_labels`. В результате этого сеть обучится сопоставлять изображения с метками. Затем мы предложим сети классифицировать изображения в `test_images` и проверим точность классификации по меткам из `test_labels`.

Теперь сконструируем сеть. Не забывайте — от вас никто не ждет, что вы поймете в этом примере все и сразу.

### **Листинг 2.2.** Архитектура сети

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Основным строительным блоком нейронных сетей является *слой* (или *уровень*), модуль обработки данных, который можно рассматривать как фильтр для данных. Он принимает некоторые данные и выводит их в более полезной форме. В частности, слои извлекают *представления* из подаваемых в них данных, которые, как мы надеемся, будут иметь больше смысла для решаемой задачи. Фактически методика глубокого обучения заключается в объединении простых слоев, реализующих некоторую форму поэтапной очистки данных. Модель глубокого обучения можно сравнить с ситом, состоящим из последовательности фильтров все более тонкой очистки данных — слоев.

В данном случае наша сеть состоит из последовательности двух слоев `Dense`, которые являются тесно связанными (их еще называют *полносвязными*) нейронными

слоями. Второй (и последний) слой — это 10-переменный слой потерь (softmax layer), возвращающий массив с 10 оценками вероятностей (в сумме дающих 1). Каждая оценка определяет вероятность принадлежности текущего изображения к одному из 10 классов цифр.

Чтобы подготовить сеть к обучению, нужно настроить еще три параметра для этапа компиляции:

- ❑ *функцию потерь*, которая определяет, как сеть должна оценивать качество своей работы на обучающих данных и, соответственно, как корректировать ее в правильном направлении;
- ❑ *оптимизатор* — механизм, с помощью которого сеть будет обновлять себя, опираясь на наблюдаемые данные и функцию потерь;
- ❑ *метрики для мониторинга на этапах обучения и тестирования* — здесь нас будет интересовать только точность (доля правильно классифицированных изображений).

Назначение функции потерь и оптимизатора мы проясним в следующих двух главах.

#### **Листинг 2.3.** Этап компиляции

```
network.compile(optimizer='rmsprop',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
```

Перед обучением мы выполним предварительную обработку данных, преобразовав их в форму, которую ожидает получить нейронная сеть, и масштабируем их так, чтобы все значения оказались в интервале [0, 1]. Исходные данные — обучающие изображения — хранятся в трехмерном массиве (60000, 28, 28) типа uint8, значениями в котором являются числа в интервале [0, 255]. Мы преобразуем его в массив (60000, 28 \* 28) типа float32 со значениями в интервале [0, 1].

#### **Листинг 2.4.** Подготовка исходных данных

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Нам также нужно закодировать метки категорий. Этот шаг подробнее объясняется в главе 3.

#### **Листинг 2.5.** Подготовка меток

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Теперь можно начинать обучение сети, для чего в случае использования библиотеки Keras достаточно вызвать метод `fit` сети — он пытается *адаптировать* (*fit*) модель под обучающие данные:

```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [=====] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=====>.....] - ETA: 1s - loss: 0.1035 - acc:
0.9692
```

В процессе обучения отображаются две величины: потери сети на обучающих данных и точность сети на обучающих данных.

В данном случае мы достигли точности 0,989 (98,9%) на обучающих данных. Теперь проверим, как модель распознает контрольный набор:

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

Точность на контрольном наборе составила 97,8 % — немного меньше, чем на тренировочном наборе. Эта разница между точностью на тренировочном и контрольном наборах демонстрирует пример *переобучения* (*overfitting*), когда модели машинного обучения показывают худшую точность на новом наборе данных по сравнению с тренировочным. Основная речь о переобучении пойдет в главе 3.

На этом мы завершаем наш первый пример — вы только что увидели, как создать и обучить нейронную сеть классификации рукописных цифр, написав меньше 20 строк кода на Python. В следующей главе я подробнее расскажу обо всех деталях, которые мы видели в этом примере, и поясню происходящее за кулисами. Вы узнаете о тензорах, объектах хранения данных в сети; операциях с тензорами, составляющими слои; градиентном спуске, позволяющем вашей сети обучаться на учебных примерах.

## 2.2. Представление данных для нейронных сетей

В предыдущем примере мы начали с данных, хранящихся в многомерных массивах NumPy, называемых также *тензорами*. Вообще говоря, все современные системы машинного обучения используют тензоры в качестве основной структуры данных. Тензоры являются фундаментальной структурой данных — настолько фундаментальной, что это отразилось на названии библиотеки Google TensorFlow. Итак, что же такое тензор?

Фактически тензор — это контейнер для данных, практически всегда числовых. Другими словами, это контейнер для чисел. Возможно, вы уже знакомы с матрицами, которые являются двумерными тензорами: тензоры — это обобщение матриц с произвольным количеством измерений (обратите внимание, что в терминологии тензоров *измерения* часто называют *осями*).

## 2.2.1. Скаляры (тензоры нулевого ранга)

Тензор, содержащий единственное число, называется *скаляром* (скалярным, или тензором нулевого ранга). В NumPy число типа `float32` или `float64` — это скалярный тензор (или скалярный массив). Определить количество осей тензора NumPy можно с помощью атрибута `ndim`; скалярный тензор имеет 0 осей (`ndim == 0`). Количество осей тензора также называют его *рангом*. Вот пример скаляра в NumPy:

```
>>> import numpy as np  
>>> x = np.array(12)  
>>> x  
array(12)  
>>> x.ndim  
0
```

## 2.2.2. Векторы (тензоры первого ранга)

Одномерный массив чисел называют *вектором*, или тензором первого ранга. Тензор первого ранга имеет единственную ось. Далее приводится пример вектора в NumPy:

```
>>> x = np.array([12, 3, 6, 14])  
>>> x  
array([12, 3, 6, 14])  
>>> x.ndim  
1
```

Этот вектор содержит пять элементов и поэтому называется *пятимерным вектором*. Не путайте пятимерные векторы с пятимерными тензорами! Пятимерный вектор имеет только одну ось и пять значений на этой оси, тогда как пятимерный тензор имеет пять осей (и может иметь любое количество значений на каждой из них). *Мерность* может обозначать или количество элементов на данной оси (как в случае с пятимерным вектором), или количество осей в тензоре (как в пятимерном тензоре), что иногда может вызывать путаницу. В последнем случае технически более корректно говорить о *тензоре пятого ранга* (ранг тензора совпадает с количеством осей), но, как бы то ни было, для тензоров используется неоднозначное обозначение: *пятимерный тензор*.

## 2.2.3. Матрицы (тензоры второго ранга)

Массив векторов — это *матрица*, или двумерный тензор. Матрица имеет две оси (часто их называют *строками* и *столбцами*). Матрицу можно представить как прямоугольную таблицу с числами. Вот пример матрицы в NumPy:

```
>>> x = np.array([[5, 78, 2, 34, 0],  
                 [6, 79, 3, 35, 1],  
                 [7, 80, 4, 36, 2]])  
>>> x.ndim  
2
```

Элементы на первой оси называют *строками*, а на второй — *столбцами*. В предыдущем примере `[5, 78, 2, 34, 0]` — это первая строка матрицы `x`, а `[5, 6, 7]` — ее первый столбец.

## 2.2.4. Тензоры третьего и высшего рангов

Если упаковать такие матрицы в новый массив, получится трехмерный тензор, который можно представить как числовой куб. Ниже приводится пример трехмерного тензора в Numpy:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
   [6, 79, 3, 35, 1],
   [7, 80, 4, 36, 2]],
  [[5, 78, 2, 34, 0],
   [6, 79, 3, 35, 1],
   [7, 80, 4, 36, 2]],
  [[5, 78, 2, 34, 0],
   [6, 79, 3, 35, 1],
   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Упаковав трехмерные тензоры в массив, вы получите четырехмерный тензор и т. д. В глубоком обучении чаще всего используются тензоры от нулевого ранга до четырехмерных, но иногда, например при обработке видеоданных, дело может дойти и до пятимерных тензоров.

## 2.2.5. Ключевые атрибуты

Тензор определяется тремя ключевыми атрибутами:

- ❑ *Количество осей (ранг)* — например, трехмерный тензор имеет три оси, а матрица — две. В библиотеках для Python, таких как Numpy, этот атрибут тензоров имеет имя `ndim`.
- ❑ *Форма* — кортеж целых чисел, описывающих количество измерений на каждой оси тензора. Например, матрица в предыдущем примере имеет форму `(3, 5)`, а трехмерный тензор имеет форму `(3, 3, 5)`. Вектор имеет форму с единственным элементом, например `(5, )`, тогда как скаляр имеет пустую форму `()`.
- ❑ *Тип данных* (обычно в библиотеках для Python ему дается имя `dtype`) — это тип данных, содержащихся в тензоре; например, тензор может иметь тип `float32`, `uint8`, `float64` и др. В редких случаях можно встретить тензоры типа `char`. Обратите внимание, что в Numpy (и в большинстве других библиотек) отсутствуют строковые тензоры, потому что тензоры хранятся в заранее выделенных, непрерывных сегментах памяти и строки, будучи сущностями с изменяющейся длиной, препятствуют использованию такой реализации.

Чтобы добавить конкретики, вернемся к данным из MNIST, которые мы обрабатывали в первом примере. Сначала загрузим набор данных MNIST:

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
Узнаем количество осей тензора train_images, обратившись к его атрибуту ndim:
>>> print(train_images.ndim)
3
```

его форму:

```
>>> print(train_images.shape)
(60000, 28, 28)
```

и тип данных, заглянув в атрибут `dtype`:

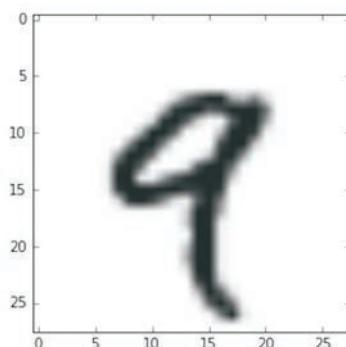
```
>>> print(train_images.dtype)
uint8
```

Итак, теперь мы знаем, что это трехмерный тензор с 8-разрядными целыми числами. Точнее, это массив с 60 000 матриц целых чисел размером  $28 \times 28$ . Каждая матрица представляет собой черно-белое изображение, где каждый элемент представляет пиксель с плотностью серого цвета в диапазоне от 0 до 255.

Попробуем отобразить четвертую цифру из этого трехмерного тензора, используя библиотеку Matplotlib (входит в состав стандартного пакета для научных вычислений) (рис. 2.2).

#### **Листинг 2.6.** Вывод четвертой цифры на экран

```
digit = train_images[4]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



**Рис. 2.2.** Четвертый образец из нашего набора данных

## 2.2.6. Манипулирование тензорами с помощью NumPy

В предыдущем примере мы *выбрали* конкретную цифру на первой оси, использовав синтаксис `train_images[i]`. Операция выбора конкретного элемента в тензоре называется *получением среза тензора*. Давайте посмотрим, какие операции получения среза тензора можно использовать с массивами NumPy.

Следующий пример извлекает цифры с 10-й до 100-й (100-я цифра не включается в срез) и помещает их в массив, имеющий форму `(90, 28, 28)`:

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
```

Это эквивалентно более подробной форме записи, в которой определяются начальный и конечный индексы среза для каждой оси тензора. Обратите внимание, что `:` эквивалентно выбору всех элементов на оси:

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

В общем случае можно получить срез между любыми двумя индексами по каждой оси тензора. Например, вот как можно выбрать пиксели из области  $14 \times 14$  в правом нижнем углу каждого изображения:

```
my_slice = train_images[:, 14:, 14:]
```

Допускается использовать и отрицательные индексы. Так же как отрицательные индексы в списках на Python, они будут откладываться от конца текущей оси. Например, вот так можно обрезать все изображения, оставив только квадрат  $14 \times 14$  пикселов в центре:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

## 2.2.7. Пакеты данных

В общем случае первая ось (ось с индексом 0, потому что нумерация начинается с 0) во всех тензорах, с которыми вам придется столкнуться в глубоком обучении, будет *осью образцов* (иногда ее называют *измерением образцов*). В примере MNIST образцы — это изображения цифр.

Кроме того, модели глубокого обучения не обрабатывают весь набор данных целиком; они разбивают его на небольшие пакеты. Если говорить конкретнее, вот один пакет из примера с изображениями цифр MNIST, имеющий размер 128:

```
batch = train_images[:128]
```

А вот следующий пакет:

```
batch = train_images[128:256]
```

А вот  $n$ -й пакет:

```
batch = train_images[128 * n:128 * (n + 1)]
```

При рассмотрении таких пакетных тензоров первую ось (ось с индексом 0) называют *осью пакетов*, или *измерением пакетов*. Эта терминология часто будет встречаться вам при работе с Keras и другими библиотеками глубокого обучения.

## 2.2.8. Практические примеры тензоров с данными

Чтобы было понятнее, перечислим несколько примеров тензоров с данными, которые могут встретиться вам в будущем. Данные, которыми вам придется манипулировать, почти всегда будут относиться к одной из следующих категорий:

- ❑ *векторные данные* — двумерные тензоры с формой (*образцы*, *признаки*);
- ❑ *временные ряды или последовательности* — трехмерные тензоры с формой (*образцы*, *метки\_времени*, *признаки*);
- ❑ *изображения* — четырехмерные тензоры с формой (*образцы*, *высота*, *ширина*, *цвет*) или с формой (*образцы*, *цвет*, *высота*, *ширина*);
- ❑ *видео* — пятимерные тензоры с формой (*образцы*, *кадры*, *высота*, *ширина*, *цвет*) или с формой (*образцы*, *кадры*, *цвет*, *высота*, *ширина*).

## 2.2.9. Векторные данные

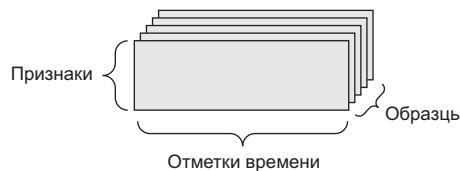
Наиболее часто встречающаяся форма данных. В таких наборах каждый образец может быть представлен вектором, а пакет, соответственно, двумерным тензором (то есть массивом векторов), где первая ось — это *ось образцов*, а вторая — *ось признаков*.

Рассмотрим два примера.

- ❑ Актуарный набор данных с информацией о людях, где для каждого человека указываются возраст, почтовый индекс и доход. Каждый человек характеризуется вектором с тремя значениями, соответственно, весь набор данных, описывающий 100 000 человек, можно сохранить в двумерном тензоре с формой (100000, 3).
- ❑ Коллекция текстовых документов, где каждый документ представлен количеством повторений каждого слова (из словаря с 20 000 наиболее употребительных слов). Каждый документ можно представить как вектор с 20 000 значений (по одному счетчику на каждое слово из словаря), соответственно, весь набор данных, описывающий 500 документов, можно сохранить в двумерном тензоре с формой (500, 20000).

## 2.2.10. Временные ряды или последовательности

Всякий раз, когда время (или понятие последовательной упорядоченности) играет важную роль в ваших данных, такие данные предпочтительнее сохранять в трехмерном тензоре с явной осью времени. Каждый образец может быть представлен как последовательность векторов (двумерных тензоров), а сам пакет данных — как трехмерный тензор (рис. 2.3).



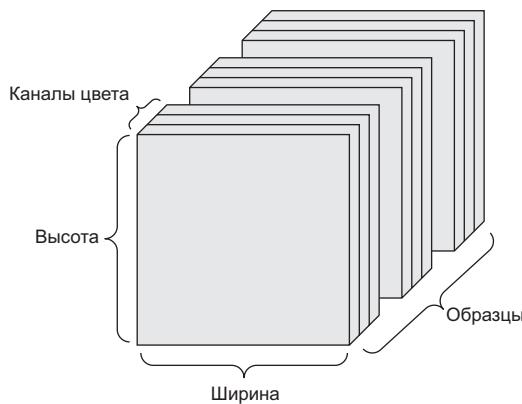
**Рис. 2.3.** Трехмерный тензор с временным рядом

В соответствии с соглашениями, ось времени всегда является второй осью (осью с индексом 1). Рассмотрим несколько примеров.

- Набор данных с ценами акций. Каждую минуту мы сохраняем текущую цену акций, а также наибольшую и наименьшую цены за минувшую минуту. То есть каждая минута представлена трехмерным вектором, весь торговый день — двумерным тензором с формой  $(390, 3)$  (где 390 — длительность торгового дня в минутах), а данные за 250 дней — трехмерным тензором с формой  $(250, 390, 3)$ . В данном случае каждый образец представляет данные за один торговый день.
- Набор данных с твитами, где каждый твит кодируется последовательностью из 280 символов из алфавита со 128 уникальными символами. В данном случае каждый символ можно закодировать как двоичный вектор со 128 элементами (содержит нули во всех элементах, кроме элемента с индексом, соответствующим номеру символа в алфавите, в который записывается 1). При такой организации каждый твит можно представить как двумерный тензор с формой  $(280, 128)$ , а набор с миллионом твитов — как тензор с формой  $(1000000, 280, 128)$ .

## 2.2.11. Изображения

Обычно изображения имеют три измерения: высоту, ширину и цвет. Даже при том, что черно-белые изображения (как в наборе данных MNIST) имеют только один канал цвета и могли бы храниться в двумерных тензорах, по соглашениям тензоры с изображениями всегда имеют три измерения, где для черно-белых изображений отводится только один канал цвета. Соответственно, пакет со 128 черно-белыми изображениями, имеющими размер  $256 \times 256$ , можно сохранить в тензоре с формой  $(128, 256, 256, 1)$ , а пакет со 128 цветными изображениями — в тензоре с формой  $(128, 256, 256, 3)$  (рис. 2.4).



**Рис. 2.4.** Четырехмерный тензор с изображениями  
(в соответствии с соглашением «канал следует первым»)

В отношении форм тензоров с изображениями существует два соглашения: соглашение *канал следует последним* (используется в TensorFlow) и соглашение *канал следует первым* (используется в Theano). Фреймворк машинного обучения TensorFlow, разработанный компанией Google, отводит для цвета последнюю ось: (*образцы, высота, ширина, цвет*). А библиотека Theano отводит для цвета ось, следующую сразу за осью пакетов: (*образцы, цвет, высота, ширина*). Если следовать соглашению, принятому в Theano, предыдущие примеры тензоров будут иметь форму  $(128, 1, 256, 256)$  и  $(128, 3, 256, 256)$ . Фреймворк Keras поддерживает оба формата.

## 2.2.12. Видео

Видеоданные — один из немногих типов данных, для хранения которых требуются пятимерные тензоры. Видео можно представить как последовательность кадров, где каждый кадр — цветное изображение. Каждый кадр можно сохранить в трехмерном тензоре (*высота, ширина, цвет*), соответственно, их последовательность можно сохранить в четырехмерном тензоре (*кадры, высота, ширина, цвет*), а пакет разных видеороликов — в пятимерном тензоре с формой (*образцы, кадры, высота, ширина, цвет*).

Например, 60-секундный видеоклип с разрешением  $144 \times 256$  и частотой 4 кадра в секунду будет состоять из 240 кадров. Для сохранения пакета из четырех таких клипов потребуется тензор с формой  $(4, 240, 144, 256, 3)$ . То есть 106 168 320 значений! Если предположить, что `dtype` тензора определен как `float32`, тогда для хранения каждого значения понадобится 32 бита, то есть для хранения всего тензора — 405 Мбайт. Мощно! Видеоролики, с которыми вам придется столкнуться в реальной жизни, намного легковеснее, потому что они не хранятся как коллекции значений типа `float32` и обычно подвергаются значительному сжатию (как, например, формат MPEG).

## 2.3. Шестеренки нейронных сетей: операции с тензорами

Так как любую компьютерную программу можно свести к небольшому набору двоичных операций с входными данными (И, ИЛИ, НЕ и др.), все преобразования, выполняемые глубокими нейронными сетями при обучении, можно свести к горстке операций с тензорами, применяемых к тензорам с числовыми данными. Например, тензоры можно складывать, перемножать и т. д.

В нашем первом примере мы создали сеть, наложив друг на друга два слоя `Dense`. В библиотеке Keras экземпляр слоя выглядит так:

```
keras.layers.Dense(512, activation='relu')
```

Этот слой можно интерпретировать как функцию, которая принимает двумерный тензор и возвращает другой двумерный тензор — новое представление исходного тензора. В данном случае функция имеет следующий вид (где  $W$  — это двумерный тензор, а  $b$  — вектор, оба значения являются атрибутами слоя):

```
output = relu(dot(W, input) + b)
```

Давайте развернем ее. Здесь у нас имеется три операции с тензорами: скалярное произведение (`dot`) исходного тензора `input` и тензора с именем `W`; сложение (+) получившегося двумерного тензора и вектора `b`; и, наконец, операция `relu`. `relu(x)` эквивалентна операции `max(x, 0)`.

### ПРИМЕЧАНИЕ

Даже при том, что в этом разделе очень часто используются выражения из линейной алгебры, вы не найдете здесь математической нотации. Как мне кажется, программисты без математического образования проще осваивают математические понятия, если они выражены короткими фрагментами кода на Python, а не математическими формулами. Поэтому мы будем везде использовать код NumPy.

### 2.3.1. Поэлементные операции

Операция `relu` и сложение — это *поэлементные операции*: операции, которые применяются к каждому элементу в тензоре по отдельности. То есть эти операции поддаются массовому распараллеливанию (*векторизации*, термин пришел из архитектуры *векторного процессора* суперкомпьютера периода 1970–1990-х). Для реализации поэлементных операций на Python можно использовать цикл `for`, как в следующем примере реализации операции `relu`:

```
def naive_relu(x):
    assert len(x.shape) == 2 ←———— Убедиться, что x — двумерный тензор NumPy
    x = x.copy() ←———— Исключить затирание исходного тензора
    for i in range(x.shape[0]):
```

```

for j in range(x.shape[1]):
    x[i, j] = max(x[i, j], 0)
return x

```

Точно так же реализуется сложение:

```

def naive_add(x, y):
    assert len(x.shape) == 2           ← Убедиться, что x и y — двумерные тензоры Numpy
    assert x.shape == y.shape
    x = x.copy()                     ← Исключить затирание исходного тензора
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x

```

Следуя тому же принципу, можно реализовать поэлементное умножение, вычитание и т. д.

При работе с массивами Numpy можно пользоваться уже готовыми, оптимизированными реализациями этих операций, доступными в виде функций из пакета Numpy, которые сами делегируют основную работу реализациям базовых подпрограмм линейной алгебры (Basic Linear Algebra Subprograms, BLAS), если они установлены (конечно же, они должны быть у вас установлены). BLAS — это комплект низкоуровневых, параллельных и эффективных процедур для вычислений с тензорами, которые обычно реализуются на Fortran или С.

Иными словами, при использовании Numpy поэлементные операции можно записывать, как показано ниже, и они будут выполняться почти мгновенно:

```

import numpy as np

z = x + y ← Поеlementnoe сложение

z = np.maximum(z, 0.) ← Поеlementnaya operatsiya relu

```

### 2.3.2. Расширение

Наша предыдущая реализация `naive_add` поддерживает только сложение двумерных тензоров с идентичными формами. Но в слое `Dense`, представленном выше, мы складывали двумерный тензор с вектором. Что происходит при сложении, когда формы складываемых тензоров различаются?

Когда это возможно и не вызывает неоднозначности, меньший тензор *расширяется* так, чтобы его новая форма соответствовала форме большего тензора. Расширение выполняется в два этапа:

1. В меньший тензор добавляются оси (называются *осями расширения*), чтобы значение его атрибута `ndim` соответствовало значению этого же атрибута большего тензора.

2. Меньший тензор копируется в эти новые оси до полного совпадения с формой большего тензора.

Рассмотрим конкретный пример. Пусть имеются тензоры  $X$  с формой  $(32, 10)$  и  $y$  с формой  $(10,)$ . Чтобы привести их в соответствие, сначала нужно добавить в тензор  $y$  первую пустую ось, чтобы он приобрел форму  $(1, 10)$ , а затем скопировать вторую ось 32 раза, чтобы в результате получился тензор  $Y$  с формой  $(32, 10)$ , где  $Y[i, :] = y$  для  $i$  в диапазоне  $\text{range}(0, 32)$ . После этого можно сложить  $X$  и  $Y$ , которые имеют одинаковую форму.

В фактической реализации новый двумерный тензор, конечно же, не создается, потому что это было бы неэффективно. Операция копирования выполняется чисто виртуально: она происходит на алгоритмическом уровне, а не в памяти. Но такое представление с копированием вектора для новой оси является полезной мысленной моделью. Вот как могла бы выглядеть наивная реализация:

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2. ←———— Убедиться, что x — двумерный тензор NumPy.
    assert len(y.shape) == 1. ←———— Убедиться, что y — вектор NumPy.
    assert x.shape[1] == y.shape[0]

    x = x.copy() ←———— Исключить затирание исходного тензора
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

Прием расширения в общем случае можно применять в поэлементных операциях с двумя тензорами, если один тензор имеет форму  $(a, b, \dots, n, n+1, \dots, m)$ , а другой — форму  $(n, n+1, \dots, m)$ . В этом случае при расширении будут добавлены оси до  $n - 1$ .

Следующий пример применяет поэлементную операцию `maximum` к двум тензорам с разными формами посредством расширения:

```
import numpy as np
x = np.random.random((64, 3, 32, 10)) ←———— x — тензор случайных чисел,
y = np.random.random((32, 10)) ←———— имеющий форму (64, 3, 32, 10)
                                            у — тензор случайных чисел,
                                            имеющий форму (32, 10)
z = np.maximum(x, y) ←———— Получившийся тензор z
                                            имеет форму (64, 3, 32, 10)
                                            аналогично x
```

### 2.3.3. Скалярное произведение тензоров

Скалярное произведение, также иногда называемое *тензорным произведением* (не путайте с поэлементным произведением), — наиболее общая и наиболее полезная

операция с тензорами. В отличие от поэлементных операций, она объединяет элементы из исходных тензоров.

Поэлементное произведение в Numpy, Keras, Theano и TensorFlow выполняется с помощью оператора `*`. Операция скалярного произведения в TensorFlow имеет иной синтаксис, но в Numpy и Keras используется простой оператор `dot`:

```
import numpy as np
z = np.dot(x, y)
```

В математике скалярное произведение обозначается точкой<sup>1</sup> (`.`):

```
z = x . y
```

Что же делает операция скалярного произведения? Для начала разберемся со скалярным произведением двух векторов,  $x$  и  $y$ :

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1           | Убедиться, что x и y — векторы Numpy
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]

    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Обратите внимание, что в результате скалярного произведения двух векторов получается скаляр и в операции могут участвовать только векторы с одинаковым количеством элементов.

Также есть возможность получить скалярное произведение матрицы  $x$  на вектор  $y$ , являющееся вектором, элементы которого — скалярные произведения строк  $x$  на  $y$ . Вот как реализуется эта операция:

```
import numpy as np

def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2           | Убедиться, что x —
    assert len(y.shape) == 1           | матрица Numpy
    assert len(y.shape) == 1           | Убедиться, что y — вектор Numpy
    assert x.shape[1] == y.shape[0]     | Первое измерение x должно совпадать
                                         | с нулевым измерением y!
                                         |
    z = np.zeros(x.shape[0])          | Эта операция вернет вектор с нулевыми
    for i in range(x.shape[0]):       | элементами, имеющий ту же форму, что и y
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

<sup>1</sup> «Точка» по-английски — «dot», отсюда название метода `dot`, реализующего скалярное произведение. — Примеч. пер.

Также можно было бы повторно использовать код, написанный прежде, подчеркнув общность произведений матрицы на вектор и вектора на вектор:

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

Обратите внимание, что, если один из двух тензоров имеет `ndim` больше 1, скалярное произведение перестает быть симметричной операцией, то есть результат `dot(x, y)` не совпадает с результатом `dot(y, x)`.

Разумеется, скалярное произведение можно распространить на тензоры с произвольным количеством осей. Наиболее часто на практике применяется скалярное произведение двух матриц. Получить скалярное произведение двух матриц, `x` и `y` (`dot(x, y)`), можно, только если `x.shape[1] == y.shape[0]`. В результате получится матрица с формой `(x.shape[0], y.shape[1])`, элементами которой являются скалярные произведения строк `x` на столбцы `y`. Вот как могла бы выглядеть простейшая реализация:

```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2 | Убедиться, что x и y — матрицы Numpy
    assert len(y.shape) == 2 | Убедиться, что x и y — матрицы Numpy
    assert x.shape[1] == y.shape[0] | Первое измерение x должно
                                    | совпадать с нулевым измерением y
    z = np.zeros((x.shape[0], y.shape[1])) | Эта операция вернет
    for i in range(x.shape[0]): | матрицу заданной формы
        for j in range(y.shape[1]): | с нулевыми элементами
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

Чтобы было понятнее, как определяется совместимость форм матриц для скалярного произведения, представьте входные и выходной тензоры, как показано на рис. 2.5.

Матрицы `x`, `y` и `z` изображены на рис. 2.5 в виде прямоугольников (буквально — таблицы элементов). Количество строк в `x` и столбцов в `y` должно совпадать, из чего следует, что ширина `x` должна совпадать с высотой `y`. Если вы будете создавать новые алгоритмы машинного обучения, вероятно, вам часто придется рисовать подобные диаграммы.

В общем случае скалярное произведение тензоров с большим числом измерений выполняется в соответствии с теми же правилами совместимости форм, как описывалось выше для случая двумерных матриц:

```
(a, b, c, d) . (d,) -> (a, b, c)
(a, b, c, d) . (d, e) -> (a, b, c, e)
```

и т. д.

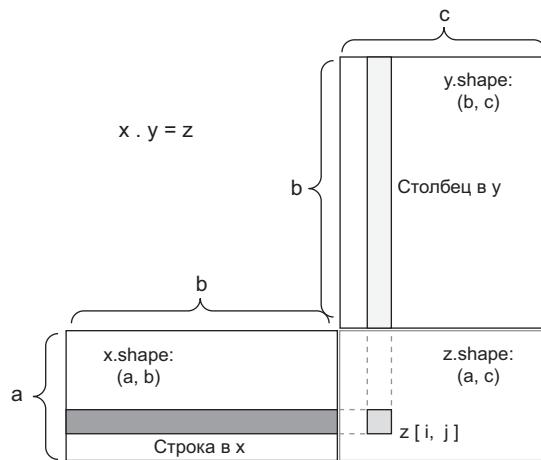


Рис. 2.5. Диаграмма скалярного произведения матриц

### 2.3.4. Изменение формы тензора

Третий вид операций с тензорами, который мы должны рассмотреть, — это *изменение формы тензора*. Эта операция не используется в слоях Dense нашей нейронной сети, но мы использовали ее, когда готовили исходные данные для передачи в сеть:

```
train_images = train_images.reshape((60000, 28 * 28))
```

Изменение формы тензора предполагает такое переупорядочение строк и столбцов, чтобы привести его форму к заданной. Разумеется, тензор с измененной формой имеет такое же количество элементов, что и исходный тензор. Чтобы было понятнее, рассмотрим несколько простых примеров:

```
>>> x = np.array([[0., 1.],
   ... [2., 3.],
   ... [4., 5.]])
>>> print(x.shape)
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

Особый случай изменения формы, который часто встречается в практике, — это **транспонирование**. Транспонирование — это такое преобразование матрицы, когда строки становятся столбцами, а столбцы — строками, то есть  $x[:, i]$  превращается в  $x[:, :, i]$ :

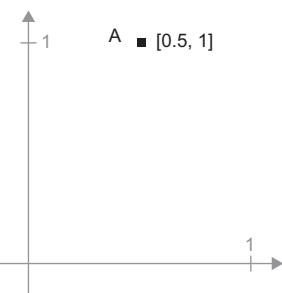
```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

### 2.3.5. Геометрическая интерпретация операций с тензорами

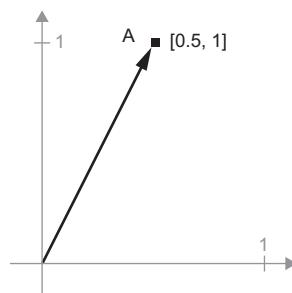
Поскольку содержимое тензоров можно интерпретировать как координаты точек в некотором геометрическом пространстве, все операции с тензорами имеют геометрическую интерпретацию. Возьмем для примера операцию сложения. Пусть имеется следующий вектор:

```
A = [0.5, 1]
```

Он определяет направление в двумерном пространстве (рис. 2.6). Векторы принято изображать в виде стрелок, соединяющих начало координат с заданной точкой, как показано на рис. 2.7.



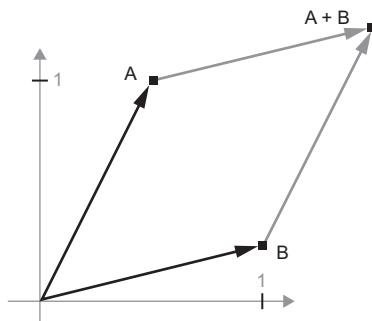
**Рис. 2.6.** Точка в двумерном пространстве



**Рис. 2.7.** Вектор в двумерном пространстве, изображенный в виде стрелки

Добавим новый вектор  $B = [1, 0.25]$  и сложим его с предыдущим. Чтобы получить результирующий вектор, представляющий сумму двух исходных векторов, достаточно перенести начало одного вектора в конец другого (рис. 2.8).

Элементарные геометрические операции, такие как аффинные преобразования, поворот, масштабирование и т. д., можно выразить в виде операций с тензорами. Например, поворот двумерного вектора на угол  $\theta$  выражается как скалярное произведение на матрицу  $R = [u, v]$  размером  $2 \times 2$ , где  $u$  и  $v$  — векторы на плоскости:  $u = [\cos(\theta), \sin(\theta)]$  и  $v = [-\sin(\theta), \cos(\theta)]$ .



**Рис. 2.8.** Геометрическая интерпретация суммы двух векторов

### 2.3.6. Геометрическая интерпретация глубокого обучения

Вы только что узнали, что нейронные сети состоят из цепочек операций с тензорами и что все эти операции, по сути, выполняют простые геометрические преобразования исходных данных. Отсюда следует, что нейронную сеть можно интерпретировать как сложное геометрическое преобразование в многомерном пространстве, реализованное в виде последовательности простых шагов.

Иногда полезно представить следующий мысленный образ в трехмерном пространстве. Вообразите два листа цветной бумаги: один лист красного цвета и другой — синего. Положите их друг на друга. Теперь скомкайте их в маленький комок. Этот мятый бумажный комок — ваши входные данные, а каждый лист бумаги — класс данных в задаче классификации. Суть работы нейронной сети (или любой другой модели машинного обучения) заключается в таком преобразовании комка бумаги, чтобы разгладить его и сделать два класса снова ясно различимыми. В глубоком обучении это реализуется как последовательность простых преобразований в трехмерном пространстве, как если бы вы производили манипуляции пальцами с бумажным комком, по одному движению за раз.



**Рис. 2.9.** Разглаживание смятого комка исходных данных

Разглаживание комка бумаги — вот что делает машинное обучение: поиск ясных представлений для сложных перемешанных данных. На данный момент у вас должно сложиться достаточно полное понимание, почему глубокое обучение преуспевает в этом: оно использует подход последовательного разложения сложных геометрических преобразований в длинную цепь простых почти так же, как по-

ступает человек, разглаживая комок бумаги. Каждый слой в глубоком обучении применяет преобразование, которое немного распутывает данные, а использование множества слоев позволяет распутывать очень сложные данные.

## 2.4. Механизм нейронных сетей: оптимизация на основе градиента

Как было показано в предыдущем разделе, каждый слой нейронной сети из нашего первого примера преобразует данные следующим образом:

```
output = relu(dot(W, input) + b)
```

В этом выражении  $W$  и  $b$  — тензоры, являющиеся атрибутами слоя. Они называются *весами*, или *обучаемыми параметрами* слоя (атрибуты `kernel` и `bias` соответствен-но). Эти веса содержат информацию, извлеченную сетью из обучающих данных.

Первоначально эти весовые матрицы заполняются небольшими случайными значениями (этот шаг называется *случайной инициализацией*). Конечно, бессмыс-ленно было бы ожидать, что `relu(dot(W, input) + b)` вернет хоть сколько-нибудь полезное представление для случайных  $W$  и  $b$ . Начальные представления не не-сут никакого смысла, но они служат начальной точкой. Далее, на основе сигнала обратной связи, происходит постепенная корректировка весов. Эта постепенная корректировка, которая также называется *обучением*, составляет суть машинного обучения.

Ниже перечислены шаги, выполняемые в так называемом *цикле обучения*, который повторяется столько раз, сколько потребуется:

1. Извлекается пакет обучающих экземпляров  $x$  и соответствующих целей  $y$ .
2. Сеть обрабатывает пакет  $x$  (этот шаг называется *прямым проходом*) и получает пакет предсказаний  $y_{pred}$ .
3. Вычисляются потери сети на пакете, дающие оценку несовпадения между  $y_{pred}$  и  $y$ .
4. Корректируются веса сети так, чтобы немного уменьшить потери на этом пакете.

В конечном итоге получается сеть, имеющая очень низкие потери на трениро-вочном наборе данных: малое несовпадение предсказаний  $y_{pred}$  с ожидаемыми целями  $y$ . Сеть «научилась» отображать входные данные в правильные конечные значения. Со стороны все это может походить на волшебство, однако если разо-брать процесс на мелкие шаги, он выглядит очень просто.

Шаг 1 не кажется сложным — это просто операция ввода/вывода. Шаги 2 и 3 — всего лишь применение нескольких операций с тензорами, и вы сможете реализо-вать эти шаги, опираясь на знания, полученные в предыдущем разделе. Наиболее сложным выглядит шаг 4: корректировка весов сети. Как по отдельным весам

в сети узнать, должен некоторый коэффициент увеличиваться или уменьшаться и насколько?

Одно из простейших решений — заморозить все веса, кроме одного, и попробовать применить разные значения этого веса. Допустим, первоначально вес имел значение 0,3. После прямого прохода потери сети составили 0,5. Теперь представьте, что после увеличения значения веса до 0,35 и повторения прямого прохода вы получили увеличение оценки потерь до 0,6, а после уменьшения веса до 0,25 — падение оценки потерь до 0,4. В данном случае похоже, что корректировка коэффициента на величину  $-0,05$  вносит свой вклад в уменьшение потерь. Этую операцию можно было бы повторить для всех весов в сети.

Но такой подход крайне неэффективен, потому что требует выполнения двух прямых проходов (что является довольно затратным) для каждого отдельного веса (которых очень много, обычно около тысячи, а иногда и до нескольких миллионов). Гораздо эффективнее опереться на тот факт, что все операции в сети *дифференцируемы*, и вычислять *градиент* потерь относительно каждого весового коэффициента в сети. После этого можно было бы просто сдвигать коэффициенты в сторону, противоположную градиенту, и тем самым уменьшать потери.

Если вы уже знаете, что означает *дифференцируемость* и что такое *градиент*, можете сразу перейти к разделу 2.4.3. Для всех остальных следующие два раздела помогут разобраться в этих понятиях.

### 2.4.1. Что такое производная?

Рассмотрим непрерывную гладкую функцию  $f(x) = y$ , отображающую вещественное число  $x$  в новое вещественное число  $y$ . Поскольку функция *непрерывна*, небольшое изменение  $x$  может дать в результате только небольшое изменение  $y$  — это вытекает из понятия непрерывности. Допустим, вы увеличили  $x$  на маленькую величину  $\epsilon_x$ : в результате  $y$  изменилось на маленькую величину  $\epsilon_y$ :

$$f(x + \epsilon_x) = y + \epsilon_y$$

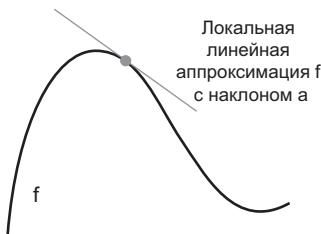
Кроме того, так как функция *гладкая* (ее кривая не имеет острых углов), когда  $\epsilon_x$  — достаточно маленькая величина отклонения, в окрестностях точки  $p$  можно аппроксимировать  $f$  линейной функцией с наклоном  $a$ , такой, что  $\epsilon_y$  станет равным  $a * \epsilon_x$ :

$$f(x + \epsilon_x) = y + a * \epsilon_x$$

Очевидно, что такая линейная аппроксимация действительна, только когда  $x$  располагается достаточно близко к точке  $p$ .

Наклон  $a$  называется *производной*  $f$  в точке  $p$ . Если  $a$  имеет отрицательное значение, значит, небольшое изменение  $x$  в окрестностях  $p$  приведет к уменьшению  $f(x)$  (как показано на рис. 2.10); а если положительное, небольшое изменение  $x$  приведет

к увеличению  $f(x)$ . Кроме того, абсолютное значение  $a$  (величина производной) сообщает, насколько большим будет это увеличение или уменьшение.



**Рис. 2.10.** Производная  $f$  в точке  $p$

Для любой дифференцируемой функции  $f(x)$  (под *дифференцируемостью* подразумевается «имеет производную»: например, гладкие непрерывные функции могут иметь производную) существует такая производная функция  $f'(x)$ , которая отображает значения  $x$  в наклон локальной линейной аппроксимации в точках  $f$ . Например, производной от  $\cos(x)$  является  $-\sin(x)$ , производной от  $f(x) = a * x - f'(x) = a$  и т. д.

Если вы пытаетесь изменить  $x$  на величину `epsilon_x`, чтобы минимизировать  $f(x)$ , и знаете производную от  $f$ , можете считать, что эту задачу вы уже решили: производная полностью описывает поведение  $f(x)$  с изменением  $x$ . Чтобы уменьшить значение  $f(x)$ , достаточно сместить  $x$  в направлении, противоположном производной.

## 2.4.2. Производная операций с тензорами: градиент

*Градиент* — это производная операции с тензором, обобщение понятия производной на функции с многомерными входными данными, то есть на функции, принимающие на входе тензоры.

Рассмотрим входной вектор  $x$ , матрицу  $W$ , цель  $y$  и функцию потерь `loss`. Вы можете с помощью  $W$  вычислить приближение к цели `y_pred` и определить потери или несоответствие, между кандидатом `y_pred` и целью `y`:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

Если входные данные  $x$  и  $y$  зафиксированы, тогда это можно интерпретировать как функцию, отображающую значения  $W$  в значения потерь:

```
loss_value = f(W)
```

Допустим, что  $W_0$  — текущее значение  $W$ . Тогда производной функции  $f$  в точке  $W_0$  будет тензор `gradient(f)(W0)` с той же формой, что и  $W$ , в котором каждый

элемент `gradient(f)(W0)[i, j]` определяет направление и величину изменения в `loss_value`, наблюдаемого при изменении `W0[i, j]`. Тензор `gradient(f)(W0)` — это градиент функции  $f(W) = loss\_value$  в `W0`.

Выше вы видели, что производную функции  $f(x)$  единственного аргумента можно интерпретировать как наклон кривой  $f$ . Аналогично `gradient(f)(W0)` можно интерпретировать как тензор, описывающий *кривизну*  $f(W)$  в окрестностях `W0`.

Соответственно, как и в случае с функцией  $f(x)$ , значение которой можно уменьшить, сместив  $x$  в направлении, противоположном производной, функцию  $f(W)$  тензора также можно уменьшить, сместив  $W$  в направлении, противоположном градиенту: например,  $W1 = W0 - step * gradient(f)(W0)$  (где `step` — небольшой по величине множитель). Это означает, что для снижения нужно идти против кривизны. Обратите внимание: множитель `step` необходим, потому что `gradient(f)(W0)` лишь аппроксимирует кривизну в окрестностях `W0`, поэтому очень нежелательно уходить слишком далеко от `W0`.

### 2.4.3. Стохастический градиентный спуск

Теоретически минимум дифференцируемой функции можно найти аналитически. Как известно, минимум функции — это точка, где производная равна 0. То есть остается только найти все точки, где производная обращается в 0, и выяснить, в какой из этих точек функция имеет наименьшее значение.

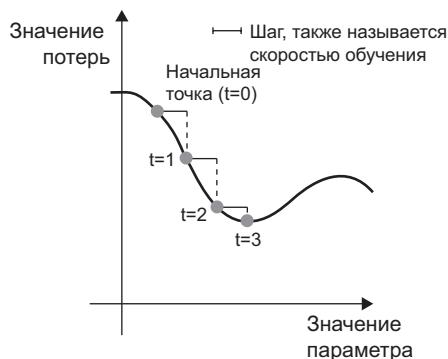
Применительно к нейронным сетям это означает аналитический поиск комбинации значений весов, при которых функция потерь будет иметь наименьшее значение. Этого можно добиться, решив уравнение  $gradient(f)(W) = 0$  для  $W$ . Это полиномиальное уравнение с  $N$  переменными, где  $N$  — количество весов в сети. Решить уравнение для случая  $N = 2$  или  $N = 3$  не составляет труда, но превращается в практически неразрешимую задачу для нейронных сетей, в которых количество параметров редко бывает меньше нескольких тысяч и часто достигает нескольких десятков миллионов.

Поэтому на практике используется алгоритм из четырех шагов, представленный в начале этого раздела: вы можете понемногу изменять параметры, опираясь на текущие значения потерь в случайном пакете данных. Поскольку функция дифференцируема, можно вычислить ее градиент, который позволяет эффективно реализовать шаг 4. Если веса изменить в направлении, противоположном градиенту, потери с каждым циклом будут понемногу уменьшаться:

1. Извлекается пакет обучающих экземпляров  $x$  и соответствующих целей  $y$ .
2. Сеть обрабатывает пакет  $x$  и получает пакет предсказаний  $y\_pred$ .
3. Вычисляются потери сети на пакете, дающие оценку несовпадения между  $y\_pred$  и  $y$ .
4. Вычисляется градиент потерь для параметров сети (*обратный проход*).

5. Параметры корректируются на небольшую величину в направлении, противоположном градиенту, например  $W = \text{step} * \text{gradient}$ , и тем самым снижаются потери.

Выглядит довольно просто! Я только что описал *стохастический градиентный спуск на небольших пакетах* (mini-batch stochastic gradient descent, minibatch SGD). Термин «*стохастический*» отражает тот факт, что каждый пакет данных выбирается случайно (в науке слово «*стохастический*» считается синонимом слова «*случайный*»). Рисунок 2.11 иллюстрирует происходящее на примере одномерных данных, когда сеть имеет только один параметр и в вашем распоряжении имеется только один обучающий образец.



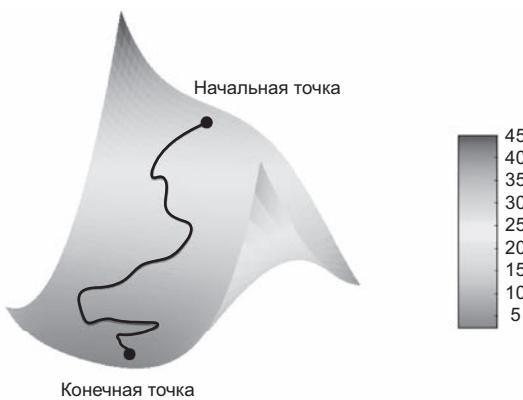
**Рис. 2.11.** Градиентный спуск вниз по одномерной кривой потерь (один обучаемый параметр)

Как можно заметить, выбор разумной величины шага `step` имеет большое значение. Если выбрать его слишком маленьким, спуск потребует большого количества итераций и может застрять в локальном минимуме. Если шаг будет слишком большим, ваши корректировки могут приобретать нецеленаправленный характер и приводить в случайные точки на кривой.

Обратите внимание, что вариант алгоритма mini-batch SGD, изображенный на рис. 2.11, в каждой итерации использует единственный образец и цель, а не весь пакет данных. Фактически это *истинный SGD* (а не *mini-batch SGD*). Однако можно пойти другим путем и использовать на каждом шаге *все* доступные данные. Эта версия алгоритма называется *пакетным стохастическим градиентным спуском* (*batch SGD*). Каждое изменение в этом случае будет более точным, но более затратным. Эффективным компромиссом между этими двумя крайностями является использование небольших пакетов.

На рис. 2.11 изображен градиентный спуск в одномерном пространстве параметров, на практике вы будете использовать градиентный спуск в пространствах с намного большим количеством измерений: каждый весовой коэффициент в нейронной

сети — это независимое измерение в пространстве, и их может быть десятки тысяч или даже миллионы. Чтобы получить представление о поверхностях потерь, представьте градиентный спуск по двумерной поверхности, как показано на рис. 2.12. Но имейте в виду, что вам не удастся мысленно представить фактический процесс обучения нейронной сети — нельзя представить 1 000 000-мерное пространство более или менее осмысленным способом. Поэтому всегда помните, что представление, полученное на таких моделях с небольшим количеством измерений, может быть не всегда точным на практике. В прошлом это часто приводило к ошибкам исследователей глубокого обучения.



**Рис. 2.12.** Градиентный спуск вниз по двумерной поверхности потерь (два обучаемых параметра)

Существует также множество вариантов стохастического градиентного спуска, которые отличаются тем, что при вычислении следующих приращений весов принимают в учет не только текущие значения градиентов, но и предыдущие приращения. Примерами могут служить такие алгоритмы, как SGD с импульсом, Adagrad, RMSProp и некоторые другие. Эти варианты известны как *методы оптимизации*, или *оптимизаторы*. В частности, внимания заслуживает идея *импульса*, которая используется во многих этих вариантах. Импульс вводится для решения двух проблем SGD: невысокой скорости сходимости и попадания в локальный минимум. Взглядите на рис. 2.13, на котором изображена кривая функции потерь для параметра сети.

Как видите, для значения данного параметра имеется *локальный минимум*: движение из этой точки влево или вправо повлечет увеличение потери. Если корректировка рассматриваемого параметра осуществляется методом градиентного спуска с маленьким шагом обучения, тогда процесс оптимизации может застрять в локальном минимуме, не найдя пути к глобальному минимуму.

Таких проблем можно избежать, если использовать идею импульса, заимствованную из физики. Вообразите, что процесс оптимизации — это маленький шарик,



**Рис. 2.13.** Локальный и глобальный минимумы

катящийся вниз по кривой потерь. Если шарик имеет достаточно высокий импульс, он не застрянет в мелком овраге и окажется в глобальном минимуме. Импульс реализуется путем перемещения шарика на каждом шаге, исходя не только из текущей величины наклона (текущего ускорения), но также из текущей скорости (набранной в результате действия силы ускорения на предыдущем шаге). На практике это означает, что приращение параметра  $w$  определяется не только по текущему значению градиента, но также по величине предыдущего приращения параметра, как показано в следующей упрощенной реализации:

```

past_velocity = 0.
momentum = 0.1 ← Постоянное значение импульса
while loss > 0.01: ← Цикл оптимизации
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum + learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)

```

#### 2.4.4. Объединение производных: алгоритм обратного распространения ошибки

В предыдущем алгоритме мы произвольно предположили, что, если функция дифференцируема, мы можем явно вычислить ее производную. На практике функция нейронной сети состоит из множества последовательных операций с тензорами, объединенных в одну цепочку, каждая из которых имеет простую, известную производную. Например, пусть есть сеть  $f$ , состоящая из трех операций с тензорами  $a$ ,  $b$  и  $c$  и весовыми матрицами  $W1$ ,  $W2$  и  $W3$ :

$$f(W1, W2, W3) = a(W1, b(W2, c(W3)))$$

Формула сообщает нам, что такую цепочку функций можно получить с использованием следующего тождества, которое называется *цепным правилом*:  $f(g(x)) = f'(g(x)) * g'(x)$ . Применение цепного правила к вычислению значений гради-

ента нейронной сети приводит к алгоритму, который называется *обратным распространением ошибки* (Backpropagation), или *обратным дифференцированием*. Обратное распространение начинается с конечного значения потери и движется в обратном направлении, от верхних слоев к нижним, применяя цепное правило для вычисления вклада каждого параметра в значение потери.

В настоящее время и еще на протяжении многих лет в будущем люди создают и будут создавать сети на основе современных фреймворков с поддержкой *символического дифференцирования*, таких как TensorFlow. Это означает, что для данной цепочки операций с известной производной они могут вычислять *функцию градиента* для цепочки (применяя цепное правило), которая отображает значения параметров сети в значения градиента. При наличии такой функции обратный проход сводится к вызову этой функции градиента. Благодаря символическому дифференцированию вам никогда не придется заниматься реализацией алгоритма обратного распространения ошибки вручную. По этой причине не будем тратить ваше время и внимание на точную формулировку алгоритма обратного распространения ошибки на страницах этой книги. Все, что вам нужно, — это хорошее понимание принципа работы оптимизации на основе градиента.

## 2.5. Оглядываясь на первый пример

Мы подошли к концу главы, и теперь вы должны неплохо представлять себе, что происходит в недрах нейронной сети. Давайте вернемся назад, к первому примеру, и рассмотрим каждую его часть в свете новых знаний, полученных в трех предыдущих разделах.

Вот наши входные данные:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Теперь вы уже знаете, что входные данные типа `float32` хранятся в тензорах Numpy, имеющих форму  $(60000, 784)$  (обучающие данные) и  $(10000, 784)$  (контрольные данные) соответственно.

Вот наша сеть:

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Сейчас вы уже знаете, что эта сеть состоит из цепочки двух слоев `Dense`, каждый из которых применяет к входным данным несколько простых операций с тензорами,

и что эти операции вовлекают весовые тензоры. Весовые тензоры, являющиеся атрибутами слоев, — это место, где запоминаются *знания*, накопленные сетью.

Вот как выглядел этап компиляции:

```
network.compile(optimizer='rmsprop',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
```

Сейчас вы уже понимаете, что `categorical_crossentropy` — это функция потерь, которая используется в качестве сигнала обратной связи для обучения весовых тензоров и которую этап обучения стремится свести к минимуму. Вы также знаете, что снижение потерь достигается за счет применения алгоритма стохастического градиентного спуска на небольших пакетах. Точные правила, управляющие конкретным применением градиентного спуска, определяются оптимизатором `rmsprop`, который передается в первом аргументе.

Наконец, вот как выглядел цикл обучения:

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Теперь вы знаете, что происходит в вызове `fit`: сеть начинает перебирать обучающие данные мини-пакетами по 128 образцов и выполняет 5 итераций (каждая итерация по всем обучающим данным называется *эпохой*). Для каждого мини-пакета сеть вычисляет градиенты весов с учетом потерь в пакете и изменяет значения весов в соответствующем направлении. В течение пяти эпох сеть выполнит 2345 изменений градиента (по 469 на эпоху), после чего потери сети окажутся достаточно низкими, чтобы эта сеть смогла классифицировать рукописные цифры с высокой точностью.

Теперь вы знаете большую часть из того, что нужно знать о нейронных сетях.

## Краткие итоги главы

- ❑ *Обучение* означает поиск комбинации параметров модели, минимизирующих функцию потерь для данного набора обучающих данных и соответствующих им целей.
- ❑ Обучение происходит путем извлечения пакетов случайных образцов данных и их целей и вычисления градиента параметров сети с учетом потерь в пакете. Затем параметры сети немного смещаются (величина смещения определяется скоростью обучения) в направлении, противоположном направлению градиента.
- ❑ Весь процесс обучения становится возможным благодаря тому обстоятельству, что нейронные сети являются цепочками дифференцируемых операций с тензорами и, следовательно, позволяют применять цепное правило для вывода функции градиента, отображающей текущие параметры и текущий пакет данных в значение градиента.

- В последующих главах вам часто будут встречаться два ключевых понятия — *функции потерь* и *оптимизаторы*. Они должны быть определены до передачи данных в сеть.
- *Функция потерь* — это величина, которую требуется свести к минимуму в ходе обучения, поэтому она должна представлять собой меру успеха для решаемой вами задачи.
- *Оптимизатор* определяет точный способ использования градиента потерь для изменения параметров: например, это может быть оптимизатор RMSProp, реализующий градиентный спуск с импульсом, и др.

# 3 Начало работы с нейронными сетями

Эта глава охватывает следующие темы:

- ✓ базовые компоненты нейронных сетей;
- ✓ введение в Keras;
- ✓ настройка рабочей станции для глубокого обучения;
- ✓ использование нейронных сетей для решения простых задач классификации и регрессии.

Цель этой главы — помочь вам начать использовать нейронные сети для решения практических задач. Здесь вы закрепите знания, приобретенные в нашем первом практическом примере в главе 2, и примените их для решения трех новых задач, охватывающих три наиболее типичных случая использования нейронных сетей: бинарной классификации, многоклассовой классификации и скалярной регрессии.

В этой главе мы подробнее рассмотрим основные компоненты нейронных сетей, с которыми познакомились в главе 2: слоями, сетями, целевыми функциями и оптимизаторами. Мы коротко ознакомимся с Keras — библиотекой глубокого обучения для Python, которую будем использовать на протяжении всей книги. Подготовим рабочую станцию для глубокого обучения, настроив TensorFlow, Keras и поддержку вычислений с GPU. Исследуем три вводных примера использования нейронных сетей для решения практических задач:

- ❑ классификацию отзывов о фильмах на положительные и отрицательные (бинарная классификация);
- ❑ классификацию новостных лент по темам (многоклассовая классификация);
- ❑ оценку стоимости дома с учетом данных о недвижимости (регрессия).

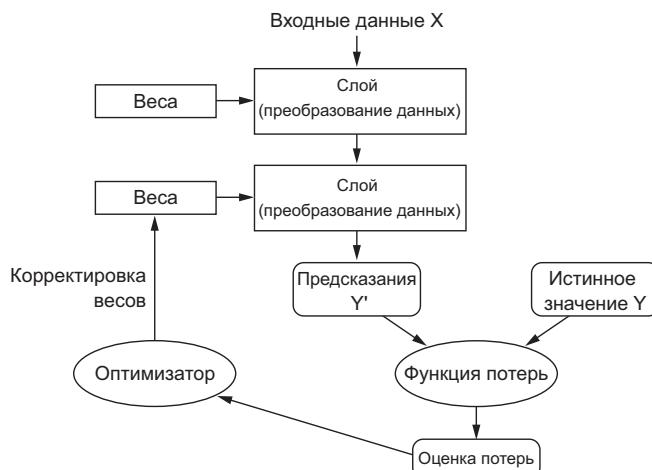
К концу этой главы вы научитесь использовать нейронные сети для решения таких задач, как классификация и регрессия по векторным данным. После этого вы будете готовы приступить к изучению более строгой теории машинного обучения в главе 4.

### 3.1. Анатомия нейронной сети

Как мы узнали в предыдущих главах, обучение нейронных сетей сосредоточено на следующих объектах:

- ❑ *слоях*, которые объединяются в сеть (или модель);
- ❑ *исходных данных* и соответствующих им *целях*;
- ❑ *функции потерь*, которая определяет сигнал обратной связи, используемый для обучения;
- ❑ *оптимизаторе*, определяющем, как происходит обучение.

Их связь можно представить, как показано на рис. 3.1: сеть состоит из слоев, которые объединяются в цепочку и отображают исходные данные в предсказания. Затем функция потерь сравнивает эти предсказания с целями и возвращает значение потери: меру соответствия предсказания, произведенного сетью, ожидаемому результату. Оптимизатор использует это значение потери для изменения весов сети.



**Рис. 3.1.** Связь между сетью, слоями, функцией потерь и оптимизатором

Познакомимся ближе со слоями, сетями, функцией потерь и оптимизаторами.

### 3.1.1. Слои: строительные блоки глубокого обучения

*Слои*, о которых рассказывалось в главе 2, являются фундаментальной структурой данных в нейронных сетях. Слой — это модуль обработки данных, принимающий на входе и возвращающий на выходе один или несколько тензоров. Некоторые слои не сохраняют состояния, но чаще это не так: *веса* слоя, один или несколько тензоров, обучаемых с применением алгоритма стохастического градиентного спуска, которые вместе хранят *знание* сети.

Разным слоям соответствуют тензоры разных форматов и разные виды обработки данных. Например, простые векторные данные, хранящиеся в двумерных тензорах с формой (*образцы, признаки*), часто обрабатываются *плотно связанными* слоями, которые также называют *полносвязными*, или *плотными*, слоями (класс `Dense` в Keras). Ряды данных хранятся в трехмерных тензорах с формой (*образцы, метки\_времени, признаки*) и обычно обрабатываются *рекуррентными* слоями, такими как `LSTM`. Изображения хранятся в четырехмерных тензорах и обычно обрабатываются двумерными сверточными слоями (`Conv2D`).

Слои можно считать кубиками LEGO глубокого обучения. Фреймворки наподобие Keras делают это сравнение еще более явным. Создание моделей глубокого обучения в Keras осуществляется путем объединения совместимых слоев в конвейеры обработки данных. Понятие *совместимости слоев* в данном случае отражает лишь тот факт, что каждый слой принимает и возвращает тензоры определенной формы. Взгляните на следующий пример:

```
from keras import layers

layer = layers.Dense(32, input_shape=(784,))
```

←

Полносвязный слой

с 32 выходными нейронами

Здесь создается слой, принимающий только двумерные тензоры, первое измерение которых равно 784 (ось 0 — измерение пакетов — не задана, поэтому допустимо любое значение). Этот слой возвращает тензор, первое измерение которого равно 32.

Другими словами, этот слой можно связать со слоем ниже, только если тот принимает двумерные векторы. Фреймворк Keras избавляет от необходимости беспокоиться о совместимости, потому что слои, добавляемые в модели, автоматически конструируются так, чтобы соответствовать форме входного слоя. Например, представьте, что вы написали следующий код:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

Второй слой создается без явного значения для аргумента `input_shape`, поэтому форма входных данных будет автоматически выведена из формы выходных данных предыдущего слоя.

### 3.1.2. Модели: сети слоев

Модель глубокого обучения является ориентированным, ациклическим графом слоев. Чаще всего на практике используется линейный стек слоев, отображающих единственный вход в единственный выход.

Однако по мере движения вперед вам встретится намного более широкий спектр топологий сетей. Вот некоторые из них:

- ❑ сети с двумя ветвями (two-branch networks);
- ❑ многоголовые сети (multihead networks);
- ❑ входные блоки (inception blocks).

Топология сети определяет *пространство гипотез*. Вспомните, как в главе 1 мы определили, что машинное обучение — это «поиск значимого представления некоторых входных данных в предопределенном пространстве возможностей с использованием сигнала обратной связи». Выбирая топологию сети, вы ограничиваете *пространство возможностей* (пространство гипотез) определенной последовательностью операций с тензорами, отображающими входные данные в выходные. Ваша задача затем — найти хороший набор значений для весовых тензоров, вовлеченных в эти операции с тензорами.

Выбор правильной архитектуры сети — это больше искусство, чем наука; и хотя есть некоторые методы и принципы, на которые можно положиться, только практика может помочь вам стать опытным архитектором нейронных сетей. В следующих нескольких главах вы познакомитесь с некоторыми принципами конструирования нейронных сетей и получите начальное представление о том, что подходит, а что не подходит для решения конкретных задач.

### 3.1.3. Функции потерь и оптимизаторы: ключи к настройке процесса обучения

После того как вы определились с архитектурой сети, требуется также выбрать еще два параметра:

- ❑ *Функцию потерь* (*целевую функцию*), возвращающую количественную оценку, которая будет минимизироваться в процессе обучения. Представляет собой меру успеха в решении стоящей задачи.
- ❑ *Оптимизатор*, определяющий, как будет изменяться сеть под воздействием функции потерь. Реализует конкретный вариант стохастического градиентного спуска (Stochastic Gradient Descent, SGD).

Нейронная сеть с несколькими выходами может иметь несколько функций потерь (по одной на выход). Однако процесс градиентного спуска должен быть основан на *единственном* скалярном значении потери, то есть для сетей с несколькими функциями потерь все потери *объединяются* (*усреднением*) в единственное скалярное значение.

Выбор правильной целевой функции для решения конкретной задачи играет очень важную роль: ваша сеть будет использовать любую возможность, чтобы минимизировать потери; поэтому, если целевая функция не полностью коррелирует с успешным решением задачи, ваша сеть в конечном итоге произведет результат, возможно, совсем не тот, что вам нужен. Представьте глупый и всемогущий ИИ, обученный методом градиентного спуска с неправильно выбранной целевой функцией: «максимизировать среднее благосостояние всех живущих людей». Чтобы упростить себе работу, такой ИИ мог бы уничтожить всех людей, кроме нескольких, и сосредоточиться на благосостоянии оставшихся, поскольку среднее благосостояние не зависит от количества оставшихся. Результат может оказаться совсем не таким, какой вы имели в виду! Просто помните, что все нейронные сети, которые вы строите, будут столь же беспощадны в минимизации функции потерь, поэтому мудро выбирайте цель, иначе вам придется столкнуться с неожиданными побочными эффектами.

К счастью для общих проблем, таких как классификация, регрессия и предсказание последовательностей, имеются простые рекомендации, которым можно следовать при выборе функции потерь. Например, для классификации в две категории можно использовать функцию бинарной перекрестной энтропии, для классификации в несколько категорий — многозначной перекрестной энтропии, для задач регрессии — среднеквадратичной ошибки, для обучения на последовательностях — ассоциативной временной классификации (Connectionist Temporal Classification, CTC) и т. д. Только сталкиваясь с действительно новыми исследовательскими задачами, вам придется разрабатывать свои целевые функции. В следующих нескольких главах мы подробно объясним, какие функции потерь следует выбирать для широкого круга типичных задач.

## 3.2. Введение в Keras

На протяжении всей книги в примерах кода будет использоваться фреймворк Keras (<https://keras.io>). Keras — это фреймворк поддержки глубокого обучения для Python, обеспечивающий удобный способ создания и обучения практически любых моделей глубокого обучения. Первоначально Keras разрабатывался для исследователей с целью дать им возможность быстро проводить эксперименты.

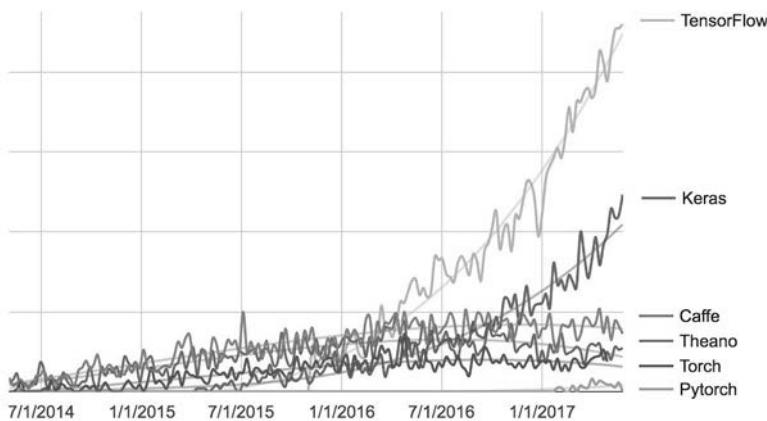
Keras обладает следующими ключевыми характеристиками:

- ❑ позволяет выполнять один и тот же код на CPU или GPU;
- ❑ имеет дружественный API, упрощающий разработку прототипов моделей глубокого обучения;

- включает в себя встроенную поддержку сверточных сетей (для распознавания образов), рекуррентных сетей (для обработки последовательностей) и всевозможных их комбинаций;
- включает в себя поддержку произвольных сетевых архитектур: моделей с множественными входами или выходами, совместное использование слоев, совместное использование моделей и т. д. Это означает, что Keras подходит для создания практически любых моделей глубокого обучения, от порождающих состязательных сетей (*generative adversarial network*) до нейронной машины Тьюринга.

Фреймворк Keras распространяется на условиях свободной лицензии и может бесплатно использоваться в коммерческих проектах. Он совместим с любыми версиями Python, от 2.7 до 3.6 (по состоянию на середину 2017 года).

Насчитывается более 200 000 пользователей Keras, начиная с академических исследователей и инженеров в стартапах и крупных компаниях и заканчивая аспирантами и любителями. Keras используется в Google, Netflix, Uber, CERN, Yelp, Square и сотнях стартапов, решающих широкий круг задач. Keras также пользуется большой популярностью среди участников состязаний по машинному обучению, проводимых сайтом Kaggle, где почти все недавние конкурсы по глубокому обучению были выиграны с использованием моделей Keras.



**Рис. 3.2.** Рост количества поисковых запросов для разных фреймворков глубокого обучения по данным Google

### 3.2.1. Keras, TensorFlow, Theano и CNTK

Keras — это библиотека уровня модели, предоставляющая высокоуровневые строительные блоки для конструирования моделей глубокого обучения. Она не реализует низкоуровневые операции, такие как манипуляции с тензорами и дифференцирование, — для этого используется специализированная и оптимизированная

библиотека поддержки тензоров. При этом Keras не полагается на какую-то одну библиотеку поддержки тензоров, а использует модульный подход (рис. 3.3); то есть к фреймворку Keras можно подключить несколько разных низкоуровневых библиотек. В настоящее время поддерживаются три такие библиотеки: TensorFlow, Theano и Microsoft Cognitive Toolkit (CNTK). В будущем Keras, скорее всего, будет расширен еще несколькими низкоуровневыми механизмами поддержки глубокого обучения.



**Рис. 3.3.** Программно-аппаратный стек поддержки глубокого обучения

TensorFlow, CNTK и Theano — это одни из ведущих платформ глубокого обучения в настоящее время. Theano (<http://deeplearning.net/software/theano>) разработана в лаборатории MILA Монреальского университета, TensorFlow ([www.tensorflow.org](http://www.tensorflow.org)) разработана в Google, а CNTK (<https://github.com/Microsoft/CNTK>) разработана в Microsoft. Любой код, использующий Keras, можно запускать с любой из этих библиотек без необходимости менять что-либо в коде: вы можете легко переключаться между ними в процессе разработки, что часто оказывается полезно, например, если одна из библиотек показывает более высокую производительность при решении данной конкретной задачи. Мы рекомендуем по умолчанию использовать библиотеку TensorFlow как наиболее распространенную, масштабируемую и высококачественную.

Используя TensorFlow (Theano или CNTK), Keras может выполнять вычисления и на CPU, и на GPU. При выполнении на CPU TensorFlow сама использует низкоуровневую библиотеку специализированных операций с тензорами, которая называется Eigen (<http://eigen.tuxfamily.org>). При выполнении на GPU TensorFlow использует оптимизированную библиотеку под названием NVIDIA CUDA Deep Neural Network (cuDNN).

### 3.2.2. Разработка с использованием Keras: краткий обзор

Вы уже видели один пример модели Keras: в примере с рукописными цифрами из набора MNIST. Вот как выглядит типичный процесс использования Keras:

1. Определяются обучающие данные: входные и целевые тензоры.
2. Определяются слои сети (*модель*), отображающие входные данные в целевые.

3. Настраивается процесс обучения выбором функции потерь, оптимизатора и некоторых параметров для мониторинга.

4. Выполняются итерации по обучающим данным вызовом метода `fit()` модели.

Модель можно определить двумя способами: с использованием класса `Sequential` (только для линейного стека слоев — наиболее популярная архитектура сетей в настоящее время) или *функционального API* (для ориентированного ациклического графа слоев, позволяющего конструировать произвольные архитектуры).

Для напоминания ниже приводится определение двухслойной модели с использованием класса `Sequential` (обратите внимание, что первому слою передается ожидаемая форма входных данных):

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

А вот та же модель, но сконструированная с применением функционального API:

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

Функциональный API позволяет манипулировать данными в тензорах, которые обрабатывает модель, и применять слои к этим тензорам, как если бы они были функциями.

### ПРИМЕЧАНИЕ

Возможности функционального API подробно описываются в главе 7. До этой главы в наших примерах мы будем использовать только класс `Sequential`.

После определения архитектуры уже неважно, используете вы модель `Sequential` или функциональный API. В обоих случаях далее выполняются одинаковые шаги.

Настройка процесса обучения производится на этапе компиляции. При этом задаются оптимизатор и функция(-и) потерь, которые должны использоваться моделью, а также все метрики для мониторинга во время обучения. Вот пример с единственной функцией потерь, которая используется чаще всего:

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='mse',
              metrics=['accuracy'])
```

Наконец, сам процесс обучения состоит в передаче массивов NumPy с входными данными (и соответствующими целевыми данными) в метод `fit()` модели, по аналогии с некоторыми другими библиотеками машинного обучения, такими как Scikit-Learn:

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

На протяжении нескольких следующих глав мы будем разбирать, какие архитектуры сетей к решению каких задач применяются, как выбрать правильную конфигурацию обучения и как настроить модель для получения желаемых результатов. В разделах 3.4, 3.5 и 3.6 мы рассмотрим три простых примера: классификации с двумя классами, классификации с несколькими классами и регрессии.

## 3.3. Настройка рабочей станции для глубокого обучения

Прежде чем приступать к разработке приложений глубокого обучения, нужно настроить рабочую станцию. Для выполнения кода, реализующего глубокое обучение, рекомендуется, хотя это и не является обязательным требованием, использовать современный графический процессор NVIDIA. Некоторые приложения, в частности обработка изображений с применением сверточных сетей и обработка последовательностей с применением рекуррентных нейронных сетей, показывают крайне низкую производительность на CPU, порой на очень быстрых многоядерных. И даже для приложений, которые вполне могут выполняться на CPU, часто выполнение на современном GPU дает прирост скорости примерно в 5–10 раз. Если у вас нет желания или возможности устанавливать GPU на свой компьютер, вы можете рассмотреть возможность проведения экспериментов на экземпляре AWS EC2 GPU или в Google Cloud Platform. Но имейте в виду, что облачные экземпляры GPU могут дорожать с течением времени.

Независимо от выбора локальной или облачной станции, лучше использовать рабочую станцию, действующую под управлением UNIX. Технически можно использовать Keras в Windows (все три библиотеки, используемые фреймворком Keras, поддерживают Windows), но мы не рекомендуем это делать. В инструкциях по установке, которые приводятся в приложении А, мы рассмотрим компьютер с ОС Ubuntu. Если вы пользуетесь Windows, просто установите Ubuntu на свой компьютер в качестве второй операционной системы. Это может показаться слишком хлопотным, однако использование Ubuntu сэкономит вам массу времени и избавит от многих проблем в будущем.

Обратите внимание на то, что для использования Keras вам нужно установить библиотеку TensorFlow, CNTK или Theano (или все три, если хотите иметь возможность переключаться между ними). В этой книге мы будем использовать TensorFlow и иногда упоминать Theano. Но мы не будем касаться библиотеки CNTK.

### 3.3.1. Jupyter Notebook: предпочтительный способ проведения экспериментов с глубоким обучением

Интерактивная оболочка Jupyter Notebook — отличный способ проведения экспериментов по глубокому обучению, и этот инструмент используется во многих примерах в этой книге. Она широко применяется в сообществах машинного обучения и науки о данных.

Блокнот (notebook, также встречается термин «тетрадь» или «ноутбук») — это файл, сгенерированный приложением Jupyter Notebook (<https://jupyter.org>), который можно редактировать в браузере. В блокнот можно вставлять код на Python и сопровождать результаты его выполнения примечаниями с возможностью форматировать текст. Создавая блокноты, можно также разбивать длительный эксперимент на несколько коротких шагов, которые выполняются независимо, что добавляет интерактивности в разработку и избавляет от необходимости повторно запускать предыдущий код, если что-то пошло не так на следующем шаге в эксперименте.

Мы рекомендуем использовать Jupyter Notebook на первых порах работы с Keras, хотя это и не является обязательным требованием: вы также можете запускать автономные сценарии на Python или выполнять код в интегрированной среде, такой как PyCharm. Все примеры в этой книге доступны в виде блокнотов с открытым исходным кодом; вы можете загрузить их с веб-сайта книги: [www.manning.com/books/deep-learning-with-python](http://www.manning.com/books/deep-learning-with-python).

### 3.3.2. Подготовка Keras: два варианта

Для практического использования мы рекомендуем следующие два варианта:

- ❑ Использовать официальные виртуальные машины EC2 Deep Learning AMI (<https://aws.amazon.com/amazonai/amis>) и выполнять эксперименты с Keras в блокнотах Jupyter Notebook на EC2. Этот вариант рекомендуется всем, у кого на локальном компьютере нет GPU. В приложении В вы найдете пошаговое руководство, в котором содержится процедура настройки.
- ❑ Установить все с нуля на локальную рабочую станцию, действующую под управлением UNIX. В этом случае вы сможете выполнять эксперименты в блокнотах Jupyter Notebook локально или запускать обычный код на Python. Этот вариант рекомендуется тем, у кого на локальном компьютере имеется высокопроизводительный NVIDIA GPU. В приложении А вы найдете пошаговое руководство, которое проведет вас через процедуру настройки рабочей станции, действующей под управлением Ubuntu.

А теперь посмотрим, на какие компромиссы придется согласиться при выборе того или иного варианта.

### 3.3.3. Запуск заданий глубокого обучения в облаке: за и против

Если у вас еще нет GPU, который можно было бы использовать для нужд глубокого обучения (одной из последних высокопроизводительных моделей NVIDIA GPU), тогда эксперименты с глубоким обучением в облаке — это простой и недорогой способ, не требующий покупки дополнительного оборудования. При использовании Jupyter Notebook работа в облаке ничем не будет отличаться от работы на локальном компьютере. По состоянию на середину 2017 года наиболее выгодным для начинающих осваивать глубокое обучение было предложение AWS EC2. В приложении B вы найдете пошаговое руководство, описывающее, как выполнять эксперименты с блокнотами Jupyter Notebook на экземпляре EC2 GPU.

Однако тем, кто планирует заниматься глубоким обучением всерьез, такой подход не годится — он не подойдет даже начинающим, которые предполагают заниматься этим дольше нескольких недель. Экземпляры EC2 слишком дороги: тип экземпляра, рекомендованный в приложении B (`p2.xlarge`, не обладающий большой вычислительной мощностью), в середине 2017 года стоил 0,90 доллара США за час. Между тем хороший GPU обойдется вам примерно в 1000–1500 долларов США. Эта цена остается стабильной, она не растет со временем даже при улучшении характеристик GPU. Если вы намерены всерьез заняться глубоким обучением, подумайте об оснащении рабочей станции одним или несколькими GPU.

Проще говоря, вариант с EC2 хорош для начального этапа. Вы можете опробовать примеры из этой книги на экземпляре EC2 GPU. Однако если вы решите всерьез заняться глубоким обучением, приобретите собственные GPU.

### 3.3.4. Выбор GPU для глубокого обучения

Если вы решили купить GPU, какой из них лучше всего выбрать? Во-первых, это должна быть модель от NVIDIA. NVIDIA — единственная компания — производитель графических процессоров, которая инвестировала большие средства в глубокое обучение, и современные фреймворки глубокого обучения могут выполняться на картах NVIDIA.

По состоянию на середину 2017 года мы рекомендуем NVIDIA TITAN Xp как лучшую карту для глубокого обучения из имеющихся на рынке. В качестве бюджетного решения можно предложить GTX 1060. Если вы читаете эти страницы в 2018 году или позже, найдите время и поищите наиболее свежие рекомендации, так как каждый год на рынке появляются все более совершенные модели.

Начиная с этого места, мы будем полагать, что у вас имеется доступ к компьютеру с установленным фреймворком Keras — и всеми его зависимостями, — предпочтительно оснащенным GPU. Перед тем как продолжить, завершите этот шаг. Прочтите пошаговые руководства в приложениях и поищите в интернете, если вам потребуется дополнительная помощь. В настоящее время нет недостатка

в руководствах по установке Keras и других инструментов, широко используемых в глубоком обучении.

Теперь мы можем погрузиться в практические примеры использования Keras.

## 3.4. Классификация отзывов к фильмам: пример бинарной классификации

Классификация по двум классам, или бинарная классификация, является едва ли не самой распространенной задачей машинного обучения. В этом примере вы научитесь классифицировать отзывы к фильмам на положительные и отрицательные, опираясь на текст отзывов.

### 3.4.1. The IMDB dataset

Вы будете работать с набором данных IMDB: множеством из 50 000 самых разных отзывов к кинолентам в интернет-базе фильмов (Internet Movie Database). Набор разбит на 25 000 обучающих и 25 000 контрольных отзывов, каждый набор на 50 % состоит из отрицательных и на 50 % из положительных отзывов.

Для чего использовать два набора — обучающий и контрольный? Потому что никогда не следует тестировать модель машинного обучения на тех же данных, которые использовались для ее обучения! Если модель прекрасно справляется с обучающими данными, это еще не значит, что она так же хорошо будет справляться с данными, которые прежде никогда не видела; и ваша задача — создать модель, качественно распознающую новые данные (вы уже знаете, к каким классам принадлежат обучающие данные, и совершенно понятно, что вам не нужна модель, классифицирующая их). Например, возможно, что ваша модель просто *запомнит* соответствия между обучающими образцами и их целями, что совершенно бесполезно для задачи предсказания по данным, которые модель никогда не видела прежде. Подробнее этот вопрос мы обсудим в следующей главе.

Подобно MNIST, набор данных IMDB поставляется в составе Keras. Он уже готов к использованию: отзывы (последовательности слов) преобразованы в последовательности целых чисел, каждое из которых определяет позицию слова в словаре.

Код в листинге 3.1 загружает набор данных (при первом запуске на ваш компьютер будет загружено примерно 80 Мбайт данных).

#### Листинг 3.1. Загрузка набора данных IMDB

```
from keras.datasets import imdb  
  
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(  
    num_words=10000)
```

Аргумент `num_words=10000` означает, что в обучающих данных будет сохранено только 10 000 слов, наиболее часто встречающихся в обучающем наборе отзывов.

Редкие слова будут отброшены. Это позволит вам работать с вектором управляющего размера.

Переменные `train_data` и `test_data` — это списки отзывов; каждый отзыв — это список индексов слов (кодированное представление последовательности слов). Переменные `train_labels` и `test_labels` — это списки нулей и единиц, где нули соответствуют *отрицательным* отзывам, а единицы — *положительным*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]
1
```

Поскольку мы ограничили себя 10 000 наиболее употребительных слов, в наборе отсутствуют индексы больше 10 000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

Чтобы вам было понятнее, ниже показано декодирование одного из отзывов в последовательность слов на английском языке:

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

Получить обратное представление словаря, отображающее индексы в слова

*word\_index* — это словарь, отображающий слова в целочисленные индексы

Декодирование отзыва. Обратите внимание, что индексы смещены на 3, потому что индексы 0, 1 и 2 зарезервированы для слов «padding» (отступ), «start of sequence» (начало последовательности) и «unknown» (неизвестно)

### 3.4.2. Подготовка данных

Нельзя передать списки целых чисел непосредственно в нейронную сеть. Поэтому мы должны преобразовать их в тензоры. Сделать это можно двумя способами:

- ❑ Привести все списки к одинаковой длине, преобразовать их в тензоры целых чисел с формой (**образцы**, **индексы\_слов**) и затем передать их в первый слой сети, способный обрабатывать такие целочисленные тензоры (слой `Embedding`, о котором подробнее мы поговорим далее в этой книге).
- ❑ Выполнить прямое кодирование списков в векторы нулей и единиц. Это может означать, например, преобразование последовательности [3, 5] в 10 000-мерный вектор, все элементы которого содержат нули, кроме элементов с индексами 3 и 5, которые содержат единицы. Затем их можно передать в первый слой сети

типа `Dense`, способный обрабатывать векторизованные данные с вещественными числами.

Мы пойдем по второму пути, с векторизованными данными, которые создадим вручную, чтобы было понятнее.

### Листинг 3.2. Кодирование последовательностей целых чисел в бинарную матрицу

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1. ← Создание матрицы с формой (len(sequences), dimension)
    return results ← Запись единицы в элемент с данным индексом

x_train = vectorize_sequences(train_data) ← Векторизованные обучающие данные
x_test = vectorize_sequences(test_data) ← Векторизованные контрольные данные
```

Вот как теперь выглядят образцы:

```
>>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.])
```

Нам также нужно векторизовать метки, что делается очень просто:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Теперь данные готовы к передаче в нейронную сеть

### 3.4.3. Конструирование сети

Входные данные представлены векторами, а метки — скалярами (единицами и нулями): это самый простой набор данных, какой можно встретить. С задачами этого вида прекрасно справляются сети, организованные как простой стек полносвязных (`Dense`) слоев с операцией активации `relu`: `Dense(16, activation='relu')`.

Аргумент `(16)`, передаваемый каждому слою `Dense`, — это число скрытых нейронов слоя. *Скрытый нейрон* (*hidden unit*) — это измерение в пространстве представлений слоя. Как рассказывалось в главе 2, каждый слой `Dense` с операцией активации `relu` реализует следующую цепочку операций с тензорами:

```
output = relu(dot(W, input) + b)
```

Наличие 16 скрытых нейронов означает, что весовая матрица `W` будет иметь форму `(input_dimension, 16)`: скалярное произведение на `W` спроецирует входные данные в 16-мерное пространство представлений (затем будет произведено сложение с вектором смещений `b` и выполнена операция `relu`). Размерность пространства представлений можно интерпретировать как «степень свободы нейронной сети при изучении внутренних представлений». Большее количество скрытых нейронов

(большая размерность пространства представлений) позволяет сети обучаться на более сложных представлениях, но при этом увеличивается вычислительная стоимость сети, что может привести к выявлению нежелательных шаблонов (шаблонов, которые могут повысить качество классификации обучающих данных, но не контрольных).

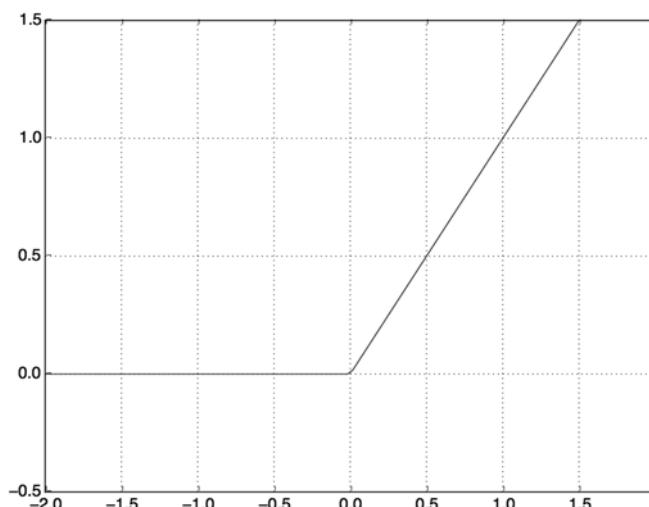
В отношении такого стека слоев `Dense` требуется принять два важных архитектурных решения:

- ❑ сколько слоев использовать;
- ❑ сколько скрытых нейронов выбрать для каждого слоя.

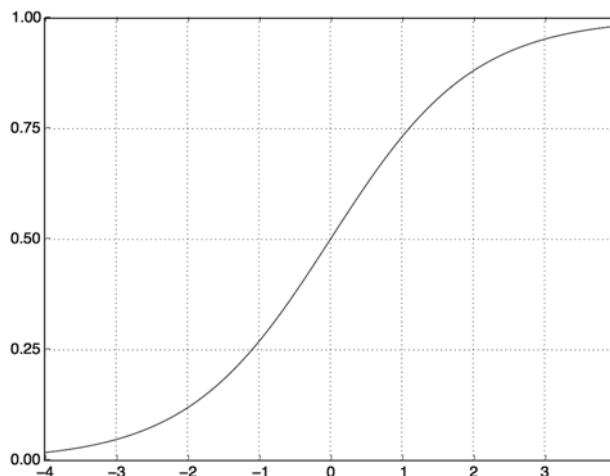
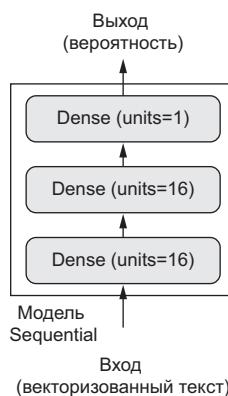
В главе 4 вы познакомитесь с формальными принципами, помогающими сделать выбор. А пока вам остается только довериться мне в следующем выборе:

- ❑ два промежуточных слоя с 16 скрытыми нейронами в каждом;
- ❑ третий слой будет выводить скалярное значение — оценку направленности текущего отзыва.

Промежуточные слои будут использовать операцию `relu` в качестве функции активации, а последний слой будет использовать сигмоидную функцию активации и выводить вероятность (оценку вероятности, между 0 и 1 того, что образец относится к классу «1», то есть насколько он близок к положительному отзыву). Функция `relu` (rectified linear unit — блок линейной ректификации) используется для преобразования отрицательных значений в ноль (рис. 3.4), а сигмоидная функция рассредоточивает произвольные значения по интервалу  $[0, 1]$  (рис. 3.5), возвращая значения, которые можно интерпретировать как вероятность.



**Рис. 3.4.** Функция блока линейной ректификации

**Рис. 3.5.** Сигмоидная функция**Рис. 3.6.** Трехслойная сеть

На рис. 3.6 показано, как выглядит сеть. Реализация этой сети с использованием Keras напоминает пример MNIST, который мы видели раньше.

### Листинг 3.3. Определение модели

```

from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
  
```

## ЧТО ТАКОЕ ФУНКЦИИ АКТИВАЦИИ И ЗАЧЕМ ОНИ НУЖНЫ?

Без функции активации, такой как `relu` (также называемой *фактором нелинейности*), слой `Dense` будет состоять из двух линейных операций — скалярного произведения и сложения:

```
output = dot(W, input) + b
```

Такой слой сможет обучаться только на линейных (аффинных) преобразованиях входных данных: пространство гипотез слоя было бы совокупностью всех возможных линейных преобразований входных данных в 16-мерное пространство. Такое пространство гипотез слишком ограниченно, и наложение нескольких слоев представлений друг на друга не приносит никакой выгоды, потому что глубокий стек линейных слоев все равно реализует линейную операцию: добавление новых слоев не расширяет пространства гипотез.

Чтобы получить доступ к более обширному пространству гипотез, дающему дополнительные выгоды от увеличения глубины представлений, необходимо применить нелинейную функцию, или функцию активации. Функция активации `relu` — самая популярная в глубоком обучении, однако на выбор имеется еще несколько функций активации с немного странными на первый взгляд именами: `prelu`, `elu` и т. д.

Наконец, нужно выбрать функцию потерь и оптимизатор. Так как перед нами стоит задача бинарной классификации и результатом работы сети является вероятность (наша сеть заканчивается одномодульным слоем с сигмоидной функцией активации), предпочтительнее использовать функцию потерь `binary_crossentropy`. Но это не единственный приемлемый выбор: можно также задействовать, например, `mean_squared_error`. Однако перекрестная энтропия обычно дает более качественные результаты, когда результатами работы моделей являются вероятности. *Перекрестная энтропия* (`crossentropy`) — это термин из области теории информации, обозначающий меру расстояния между распределениями вероятностей, или в данном случае — между фактическими данными и предсказаниями.

На этом шаге мы настраиваем модель оптимизатором `rmsprop` и функцией потерь `binary_crossentropy`. Обратите внимание, что мы также задали мониторинг точности во время обучения.

### Листинг 3.4. Компиляция модели

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Оптимизатор, функция потерь и метрики передаются в виде строковых значений, что становится возможным, поскольку `rmsprop`, `binary_crossentropy` и `accuracy` являются частью Keras. Иногда бывает желательно настроить параметры оптимизатора или передать свою функцию потерь или метрик. Первую задачу можно решить путем передачи в аргументе `optimizer` экземпляра класса оптимизатора, как показано в листинге 3.5, а последнюю — путем передачи в аргументе `loss` и (или) `metrics` объекта функции, как показано в листинге 3.6.

**Листинг 3.5.** Настройка оптимизатора

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

**Листинг 3.6.** Использование нестандартных функций потерь и метрик

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

### 3.4.4. Проверка решения

Чтобы проконтролировать точность модели во время обучения на данных, которые она прежде не видела, создадим проверочный набор, выбрав 10 000 образцов из оригинального набора обучающих данных.

**Листинг 3.7.** Создание проверочного набора

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Теперь проведем обучение модели в течение 20 эпох (выполнив 20 итераций по всем образцам в тензорах `x_train` и `y_train`) пакетами по 512 образцов. В то же время будем следить за потерями и точностью на 10 000 отложенных образцов. Для этого достаточно передать проверочные данные в аргументе `validation_data`.

**Листинг 3.8.** Обучение модели

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(partial_x_train,
                     partial_y_train,
                     epochs=20,
                     batch_size=512,
                     validation_data=(x_val, y_val))
```

При использовании CPU на каждую эпоху будет потрачено менее 2 секунд — а все обучение закончится через 20 секунд. В конце каждой эпохи обучение приостанавливается, потому что модель вычисляет потерю и точность на 10 000 образцах проверочных данных.

Обратите внимание на то, что вызов `model.fit()` возвращает объект `History`. Этот объект имеет поле `history` — словарь с данными обо всем происходившем в процессе обучения. Заглянем в него:

```
>>> history_dict = history.history
>>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

Словарь содержит четыре элемента — по одному на метрику, — за которыми осуществлялся мониторинг в процессе обучения и проверки. В следующих двух листингах используется библиотека `Matplotlib` для вывода графиков потерь (рис. 3.7), а также графиков точности на этапах обучения и проверки (рис. 3.8). Имейте в виду, что у вас результаты могут несколько различаться, что обусловлено различием в случайных числах, использовавшихся для инициализации сети.

### Листинг 3.9. Формирование графиков потерь на этапах обучения и проверки

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, loss_values, 'bo', label='Training loss') ←
plt.plot(epochs, val_loss_values, 'b', label='Validation loss') ←
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend() → «b» означает
→ «solid blue line» —
→ «сплошная синяя
линия»
plt.show()
```

«bo» означает  
«blue dot» —  
«синяя точка»

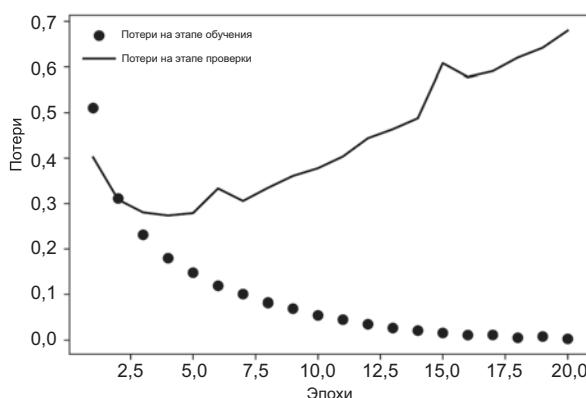


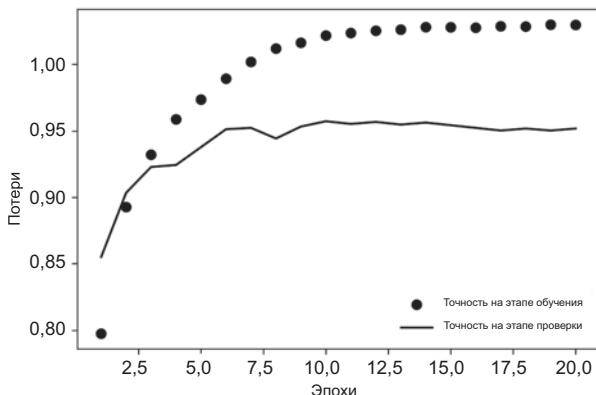
Рис. 3.7. Потери на этапах обучения и проверки

**Листинг 3.10.** Формирование графиков точности на этапах обучения и проверки

```
plt.clf() ← Очистить рисунок
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



**Рис. 3.8.** Точность на этапах обучения и проверки

Как видите, на этапе обучения потери снижаются с каждой эпохой, а точность растет. Именно такое поведение ожидается от оптимизации градиентным спуском: величина, которую вы пытаетесь минимизировать, должна становиться все меньше с каждой итерацией. Но это не относится к потерям и точности на этапе проверки: похоже, что они достигли пика в четвертую эпоху. Это пример того, о чем мы предупреждали выше: модель, показывающая хорошие результаты на обучающих данных, не обязательно будет показывать такие же хорошие результаты на данных, которые не видела прежде. Выражаясь точнее, в данном случае наблюдается *переобучение*: после второй эпохи произошла чрезмерная оптимизация на обучающих данных, и в результате получилось представление, характерное для обучающих данных, не обобщающее данные за пределами обучающего набора.

В данном случае для предотвращения переобучения можно прекратить обучение после третьей эпохи. Вообще говоря, есть целый спектр приемов, ослабляющих эффект переобучения, которые мы рассмотрим в главе 4.

А теперь обучим новую сеть с нуля в течение четырех эпох и затем оценим получившийся результат на контрольных данных.

#### Листинг 3.11. Обучение новой модели с нуля

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

Конечные результаты:

```
>>> results
[0.2929924130630493, 0.8832799999999995]
```

Это простейшее решение позволило достичь точности 88 %. При использовании же самых современных подходов точность может доходить до 95 %.

### 3.4.5. Использование обученной сети для предсказаний на новых данных

После обучения сети ее можно использовать для решения практических задач. Например, попробуем предсказать вероятность того, что отзывы будут положительными, с помощью метода `predict`:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

Как видите, сеть уверена в одних образцах (0,99 или выше или 0,01 или ниже), но не так уверена в других (0,6; 0,4).

### 3.4.6. Дальнейшие эксперименты

Следующие эксперименты помогут вам убедиться, что выбор именно таких параметров архитектуры сети был достаточно разумным, хотя место для улучшения все же остается.

- ❑ В данном примере использовались два скрытых слоя. Попробуйте использовать один или три и посмотрите, как это повлияет на точность на этапах обучения и проверки.
- ❑ Попробуйте использовать слои с большим или с меньшим количеством скрытых нейронов: 32 нейрона, 64 нейрона и т. д.
- ❑ Попробуйте вместо `binary_crossentropy` использовать функцию потерь `mse`.
- ❑ Попробуйте вместо `relu` использовать функцию активации `tanh` (она была популярна на заре становления нейронных сетей).

### 3.4.7. Подведение итогов

Вот какие выводы вы должны сделать из этого примера:

- ❑ Обычно исходные данные приходится подвергать некоторой предварительной обработке, чтобы передать их в нейронную сеть в виде тензоров. Последовательности слов можно преобразовать в бинарные векторы, но существуют также другие варианты.
- ❑ Стек слоев `Dense` с функцией активации `relu` способен решать широкий круг задач (включая классификацию эмоциональной окраски), и вы, вероятно, чаще всего будете использовать именно эту комбинацию.
- ❑ В задаче бинарной классификации (с двумя выходными классами) в конце вашей нейросети должен находиться слой `Dense` с одним нейроном и функцией активации `sigmoid`: результатом работы сети должно быть скалярное значение в диапазоне между 0 и 1, представляющее собой вероятность.
- ❑ С таким скалярным результатом, получаемым с помощью сигмоидной функции, в задачах бинарной классификации следует использовать функцию потерь `binary_crossentropy`.
- ❑ В общем случае оптимизатор `rmsprop` является наиболее подходящим выбором для любого типа задач. Одной головной болью меньше для вас.
- ❑ По мере улучшения на обучающих данных нейронные сети рано или поздно начинают переобучаться, демонстрируя ухудшение результатов на данных, которые они прежде не видели. Поэтому всегда контролируйте качество работы сети на данных не из обучающего набора.

## 3.5. Классификация новостных лент: пример классификации в несколько классов

В предыдущем разделе вы увидели, как можно классифицировать векторы входных данных на два взаимоисключающих класса с использованием полносвязной нейронной сети. Но как быть, если количество классов больше двух?

В этом разделе мы создадим сеть для классификации новостных лент агентства Reuters на 46 взаимоисключающих тем. Так как теперь количество классов больше двух, эта задача относится к категории задач *многоклассовой классификации*; и, поскольку каждый экземпляр данных должен быть отнесен только к одному классу, эта задача является примером *однозначной многоклассовой классификации*. Если бы каждый экземпляр данных мог принадлежать нескольким классам (в данном случае темам), эта задача была бы примером *многозначной многоклассовой классификации*.

### 3.5.1. Набор данных Reuters

Мы будем работать с набором данных Reuters — выборкой новостных лент и их тем, опубликовавшихся агентством Reuters в 1986 году. Это простой набор данных, широко используемых для классификации текста. Существует 46 разных тем; некоторые темы более широко представлены, некоторые — менее, но для каждой из них в обучающем наборе имеется не менее 10 примеров.

Подобно IMDB и MNIST, набор данных Reuters поставляется в составе Keras. Давайте заглянем в него.

#### Листинг 3.12. Загрузка данных Reuters

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)
```

По аналогии с примером IMDB, аргумент `num_words=10000` ограничивает данные 10 000 наиболее часто встречающимися словами.

Всего у нас имеется 8982 обучающих и 2246 контрольных примеров:

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

По аналогии с отзывами в базе данных IMDB, каждый пример — это список целых чисел (индексов слов):

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Вот как можно декодировать индексы в слова, если это представляет для вас интерес.

**Листинг 3.13.** Декодирование новостей обратно в текст

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

← Обратите внимание, что индексы смещены на 3, потому что индексы 0, 1 и 2 зарезервированы для слов «padding» (отступ), «start of sequence» (начало последовательности) и «unknwon» (неизвестно)

Метка, определяющая класс примера, — это целое число между 0 и 45 — индекс темы:

```
>>> train_labels[10]
3
```

### 3.5.2. Подготовка данных

Для векторизации данных можно повторно использовать код из предыдущего примера.

**Листинг 3.14.** Кодирование данных

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data) ← Векторизованные обучающие данные
x_test = vectorize_sequences(test_data) ← Векторизованные контрольные данные
```

Векторизовать метки можно одним из двух способов: сохранить их в тензоре целых чисел или использовать прямое кодирование. Прямое кодирование (one-hot encoding) широко используется для форматирования категорий и также называется *кодированием категорий* (categorical encoding). Более подробно прямое кодирование объясняется в разделе 6.1. В данном случае прямое кодирование меток заключается в конструировании вектора с нулевыми элементами со значением 1 в элементе, индекс которого соответствует индексу метки. Например:

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

one_hot_train_labels = to_one_hot(train_labels)
one_hot_test_labels = to_one_hot(test_labels)
```

Следует отметить, что этот способ уже реализован в Keras, как мы видели в примере MNIST:

```
from keras.utils.np_utils import to_categorical
one_hot_train_labels = to_categorical(train_labels) ← Векторизованные
one_hot_test_labels = to_categorical(test_labels) ← обучающие данные
                                            Векторизованные
                                            контрольные данные
```

### 3.5.3. Конструирование сети

Задача классификации по темам напоминает предыдущую задачу классификации отзывов: в обоих случаях мы пытаемся классифицировать короткие фрагменты текста. Но в данном случае количество выходных классов увеличилось с 2 до 46. Размерность выходного пространства теперь намного больше.

В стеке слоев `Dense`, как в предыдущем примере, каждый слой имеет доступ только к информации, предоставленной предыдущим слоем. Если один слой отбросит какую-то информацию, важную для решения задачи классификации, последующие слои не смогут восстановить ее: каждый слой может стать узким местом для информации. В предыдущем примере мы использовали 16-мерные промежуточные слои, но 16-мерное пространство может оказаться слишком ограниченным для классификации на 46 разных классов: такие малоразмерные слои могут сыграть роль «бутылочного горлышка» для информации, не пропуская важные данные.

По этой причине в данном примере мы будем использовать слои с большим количеством измерений. Давайте выберем 64 нейрона.

#### Листинг 3.15. Определение модели

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

Отметим еще две особенности этой архитектуры:

- ❑ Сеть завершается слоем `Dense` с размером 46. Это означает, что для каждого входного образца сеть будет выводить 46-мерный вектор. Каждый элемент этого вектора (каждое измерение) представляет собой отдельный выходной класс.
- ❑ Последний слой использует функцию активации `softmax`. Мы уже видели этот шаблон в примере MNIST. Он означает, что сеть будет выводить распределение вероятностей по 46 разным классам — для каждого образца на входе сеть будет возвращать 46-мерный вектор, где `output[i]` — вероятность принадлежности образца классу `i`. Сумма 46 элементов всегда будет равна 1.

Лучшим вариантом в данном случае является использование функции потерь `categorical_crossentropy`. Она определяет расстояние между распределениями вероятностей: в данном случае между распределением вероятности на выходе сети и истинным распределением меток. Минимизируя расстояние между этими двумя распределениями, мы учим сеть выводить результат, максимально близкий к истинным меткам.

**Листинг 3.16.** Компиляция модели

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

### 3.5.4. Проверка решения

Для контроля точности модели создадим проверочный набор, выбрав 1000 образцов из набора обучающих данных.

**Листинг 3.17.** Создание проверочного набора

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Теперь проведем обучение модели в течение 20 эпох.

**Листинг 3.18.** Обучение модели

```
history = model.fit(partial_x_train,
                     partial_y_train,
                     epochs=20,
                     batch_size=512,
                     validation_data=(x_val, y_val))
```

И наконец, выведем графики кривых потерь и точности (рис. 3.9 и 3.10).

**Листинг 3.19.** Формирование графиков потерь на этапах обучения и проверки

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

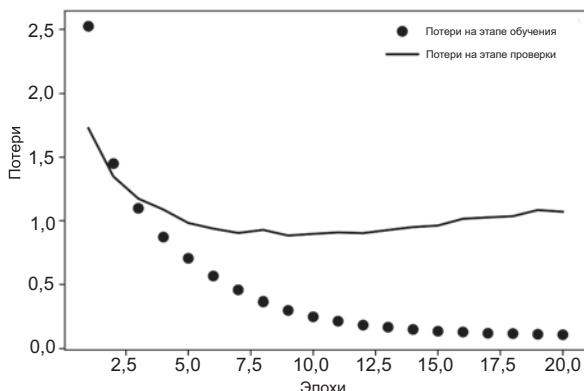


Рис. 3.9. Потери на этапах обучения и проверки

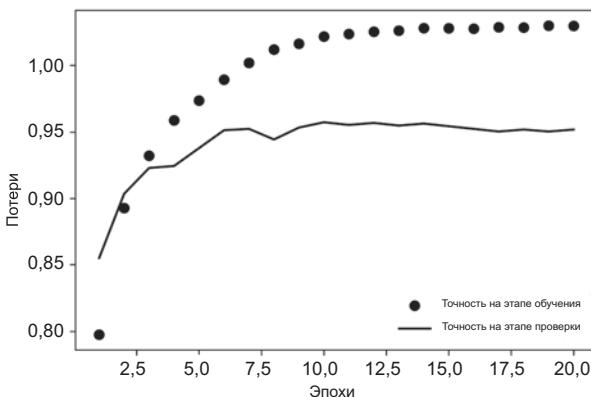


Рис. 3.10. Точность на этапах обучения и проверки

**Листинг 3.20.** Формирование графиков точности на этапах обучения и проверки

```
plt.clf() ← Очистить рисунок
acc = history.history['acc']
val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

Переобучение сети наступает в девятой эпохе. Давайте теперь обучим новую сеть до девятой эпохи и затем оценим получившийся результат на контрольных данных.

**Листинг 3.21.** Обучение новой модели с нуля

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(partial_x_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(x_val, y_val))

results = model.evaluate(x_test, one_hot_test_labels)
```

Конечные результаты:

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

Это решение достигло точности ~80 %. Со сбалансированной задачей бинарной классификации точность чисто случайного классификатора составила бы 50 %. Однако в данном случае она близка к 19 %, то есть результат получился весьма неплохим, по крайней мере в сравнении со случайнм решением:

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> float(np.sum(hits_array)) / len(test_labels)
0.18655387355298308
```

### 3.5.5. Предсказания на новых данных

Теперь можно убедиться в том, что метод `predict` модели возвращает распределение вероятностей по всем 46 темам. Давайте сгенерируем предсказания для всех контрольных данных.

**Листинг 3.22.** Получение предсказаний для новых данных

```
predictions = model.predict(x_test)
```

Каждый элемент в `predictions` — это вектор с длиной 46:

```
>>> predictions[0].shape
(46,)
```

Сумма коэффициентов этого вектора равна 1:

```
>>> np.sum(predictions[0])
1.0
```

Наибольший элемент, элемент с наибольшей вероятностью, — это предсказанный класс:

```
>>> np.argmax(predictions[0])
4
```

### 3.5.6. Другой способ обработки меток и потерь

Выше упоминалось, что метки также можно было бы преобразовать в тензор целых чисел, как показано ниже:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

Единственное, что изменилось в данном случае, — функция потерь. В листинге 3.21 использовалась функция потерь `categorical_crossentropy`, предполагающая, что метки получены методом кодирования категорий. С целочисленными метками следует использовать функцию `sparse_categorical_crossentropy`:

```
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

С математической точки зрения эта новая функция потерь равнозначна функции `categorical_crossentropy`; ее отличает только интерфейс.

### 3.5.7. Важность использования достаточно больших промежуточных слоев

Выше уже говорилось, что не следует использовать слои, в которых значительно меньше 46 скрытых нейронов, потому что результат является 46-мерным. Теперь давайте посмотрим, что получится, если образуется узкое место для информации из-за промежуточных слоев с размерностями намного меньше 46, например четырехмерных.

Теперь сеть показывает точность ~71 % — абсолютное падение составило 8 %. Это падение в основном обусловлено попыткой сжать большой объем информации (достаточной для восстановления гиперплоскостей, разделяющих 46 классов) в промежуточное пространство со слишком малой размерностью. Сети удалось вместить большую часть необходимой информации в эти четырехмерные представления, но не всю.

**Листинг 3.23.** Модель с узким местом для информации

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

### 3.5.8. Дальнейшие эксперименты

- ❑ Попробуйте использовать слои с большим или меньшим числом измерений: 128, 32 и т. д.
- ❑ Мы использовали два скрытых слоя. Теперь попробуйте использовать один слой или три.

### 3.5.9. Подведение итогов

Вот какие выводы вы должны сделать из этого примера:

- ❑ Если вы пытаетесь классифицировать образцы данных по  $N$  классам, сеть должна завершаться слоем `Dense` размера  $N$ .
- ❑ В задаче однозначной многоклассовой классификации заключительный слой сети должен иметь функцию активации `softmax`, чтобы он мог выводить распределение вероятностей между  $N$  классами.
- ❑ Для решения подобных задач почти всегда следует использовать функцию потерь `categorical_crossentropy`. Она минимизирует расстояние между распределениями вероятностей, выводимыми сетью, и истинными распределениями целей.
- ❑ Метки в многоклассовой классификации можно обрабатывать двумя способами:
  - Кодировать метки с применением метода кодирования категорий (также известного как прямое кодирование) и использовать функцию потерь `categorical_crossentropy`.
  - Кодировать метки как целые числа и использовать функцию потерь `sparse_categorical_crossentropy`.

- Когда требуется классифицировать данные относительно большого количества категорий, следует предотвращать появление в сети узких мест для информации из-за слоев с недостаточно большим количеством измерений.

## 3.6. Предсказание цен на дома: пример регрессии

В двух предыдущих примерах мы познакомились с задачами классификации, цель которых состояла в предсказании одной дискретной метки для образца входных данных. Другим распространенным типом задач машинного обучения является *регрессия*, которая заключается в предсказании не дискретной метки, а значения на непрерывной числовой прямой: например, предсказание температуры воздуха на завтра по имеющимся метеорологическим данным или предсказание времени завершения программного проекта по его спецификациям.

### ПРИМЕЧАНИЕ

Не путайте *регрессию* с алгоритмом *логистической регрессии*. Как ни странно, логистическая регрессия не является регрессионным алгоритмом — это алгоритм классификации.

### 3.6.1. Набор данных с ценами на жилье в Бостоне

Мы попытаемся предсказать медианную цену на дома в пригороде Бостона в середине 1970-х по таким данным о пригороде того времени, как уровень преступности, ставка местного имущественного налога и т. д. Набор данных, который нам предстоит использовать, имеет интересное отличие от двух предыдущих примеров. Он содержит относительно немного образцов данных: всего 506, разбитых на 404 обучающих и 102 контрольных образца. И каждый *признак* во входных данных (например, уровень преступности) имеет свой масштаб. Например, некоторые признаки являются пропорциями и имеют значения между 0 и 1, другие — между 1 и 12 и т. д.

#### Листинг 3.24. Загрузка набора данных для Бостона

```
from keras.datasets import boston_housing  
  
(train_data, train_targets), (test_data, test_targets) =  
    boston_housing.load_data()
```

Посмотрим на данные:

```
>>> train_data.shape  
(404, 13)  
>>> test_data.shape  
(102, 13)
```

Как видите, у нас имеются 404 обучающих и 102 контрольных образца, каждый с 13 числовыми признаками, такими как уровень преступности, среднее число комнат в доме, удаленность от центральных дорог и т. д.

Цели — медианные значения цен на дома, занимаемые собственниками, в тысячах долларов:

```
>>> train_targets  
[ 15.2, 42.3, 50. ... 19.4, 19.4, 29.1]
```

Цены в основной массе находятся в диапазоне от 10 000 до 50 000 долларов США. Если вам покажется, что это недорого, не забывайте, что это цены середины 1970-х и в них не были внесены поправки на инфляцию.

### 3.6.2. Подготовка данных

Было бы проблематично передать в нейронную сеть значения, имеющие самые разные диапазоны. Сеть, конечно, сможет автоматически адаптироваться к таким разнородным данным, однако это усложнит обучение. На практике к таким данным принято применять нормализацию: для каждого признака во входных данных (столбца в матрице входных данных) из каждого значения вычитается среднее по этому признаку, и разность делится на стандартное отклонение, в результате признак центрируется по нулевому значению и имеет стандартное отклонение, равное единице. Такую нормализацию легко выполнить с помощью NumPy.

#### Листинг 3.25. Нормализация данных

```
mean = train_data.mean(axis=0)  
train_data -= mean  
std = train_data.std(axis=0)  
train_data /= std  
  
test_data -= mean  
test_data /= std
```

Обратите внимание на то, что величины, используемые для нормализации контрольных данных, вычисляются с использованием обучающих данных. Никогда не следует использовать в работе какие-либо значения, вычисленные по контрольным данным, даже для таких простых шагов, как нормализация данных.

### 3.6.3. Конструирование сети

Из-за небольшого количества образцов мы будем использовать очень маленькую сеть с двумя четырехмерными промежуточными слоями. Вообще говоря, чем меньше обучающих данных, тем скорее наступит переобучение, а использование маленькой сети — один из способов борьбы с ним.

**Листинг 3.26.** Определение модели

```

from keras import models
from keras import layers

def build_model():
    model = models.Sequential() ←
        Поскольку нам потребуется несколько
        экземпляров одной и той же модели, мы
        определили функцию для ее создания
    model.add(layers.Dense(64, activation='relu',
        input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model

```

Сеть заканчивается одномерным слоем, не имеющим функции активации (это линейный слой). Это типичная конфигурация для скалярной регрессии (целью которой является предсказание одного значения на непрерывной числовой прямой). Применение функции активации могло бы ограничить диапазон выходных значений: например, если в последнем слое применить функцию активации `sigmoid`, сеть обучилась бы предсказывать только значения из диапазона между 0 и 1. В данном случае, с линейным последним слоем, сеть способна предсказывать значения из любого диапазона.

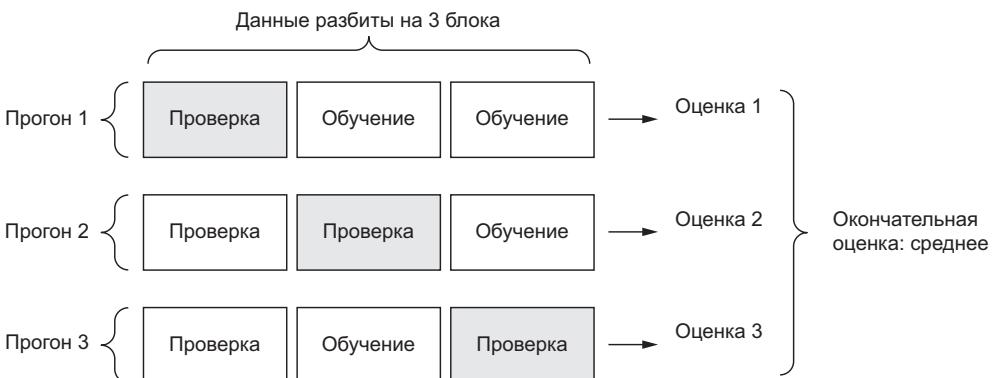
Обратите внимание на то, что сеть компилируется с функцией потерь `mse` — *mean squared error* (среднеквадратичная ошибка), вычисляющей квадрат разности между предсказанными и целевыми значениями. Эта функция широко используется в задачах регрессии.

Мы также включили новый параметр в мониторинг на этапе обучения: `mae` — *mean absolute error* (средняя абсолютная ошибка). Это абсолютное значение разности между предсказанными и целевыми значениями. Например, значение МАЕ, равное 0,5, в этой задаче означает, что в среднем прогнозы отклоняются на 500 долларов США.

### 3.6.4. Оценка решения методом перекрестной проверки по К блокам

Чтобы оценить качество сети в ходе корректировки ее параметров (таких, как количество эпох обучения), можно разбить исходные данные на обучающий и проверочный наборы, как это делалось в предыдущих примерах. Однако так как у нас и без того небольшой набор данных, проверочный набор получился бы слишком маленьким (скажем, что-нибудь около 100 образцов). Как следствие, оценки при проверке могут сильно меняться в зависимости от того, какие данные попадут в проверочный и обучающий наборы: оценки при проверке могут иметь слишком большой разброс. Это не позволит надежно оценить качество модели.

Лучшей практикой в таких ситуациях является применение *перекрестной проверки по K блокам* (K-fold cross-validation), как показано на рис. 3.11. Суть ее заключается в разделении доступных данных на K блоков (обычно K = 4 или 5), создании K идентичных моделей и обучении каждой на K–1 блоках с оценкой по оставшимся блокам. По полученным K оценкам вычисляется среднее значение, которое принимается как оценка модели. В коде такая проверка реализуется достаточно просто.



**Рис. 3.11.** Перекрестная проверка по трем блокам

### Листинг 3.27. Перекрестная проверка по K блокам

```
import numpy as np
```

```

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []

for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples] ←
    val_targets = train_targets[i * num_val_samples: (i + 1) ←
                                * num_val_samples]

    partial_train_data = np.concatenate( ←
        [train_data[:i * num_val_samples], ←
         train_data[(i + 1) * num_val_samples:], ←
         axis=0]) ←
    partial_train_targets = np.concatenate( ←
        [train_targets[:i * num_val_samples], ←
         train_targets[(i + 1) * num_val_samples:], ←
         axis=0])

    model = build_model() ←
    
```

Подготовка проверочных  
данных: данных из блока  
с номером k

Подготовка обучающих данных:  
данных из остальных блоков

Конструирование модели Keras (уже скомпилированной)

```

model.fit(partial_train_data, partial_train_targets,
          epochs=num_epochs, batch_size=1, verbose=0)
→ val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
all_scores.append(val_mae)

Оценка модели
по проверочным данным

```

Обучение модели (в режиме без вывода сообщений, verbose = 0)

Выполнив этот код с num\_epochs = 100, мы получили следующие результаты:

```

>>> all_scores
[2.588258957792037, 3.1289568449719116, 3.1856116051248984, 3.0763342615401386]
>>> np.mean(all_scores)
2.9947904173572462

```

Разные прогоны действительно показывают разные оценки, от 2,6 до 3,2. Средняя (3,0) выглядит более достоверно, чем любая из оценок отдельных прогонов, — в этом главная ценность перекрестной проверки по K блокам. В данном случае средняя ошибка составила 3000 долларов, что довольно много, если вспомнить, что цены колеблются в диапазоне от 10 000 до 50 000 долларов.

Попробуем увеличить время обучения сети до 500 эпох. Чтобы получить информацию о качестве обучения модели в каждую эпоху, изменим цикл обучения и добавим сохранение оценки проверки перед началом эпохи.

### Листинг 3.28. Сохранение оценки проверки перед каждым прогоном

```

num_epochs = 500
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples] ←
    val_targets = train_targets[i * num_val_samples:
                                (i + 1) * num_val_samples]

    partial_train_data = np.concatenate( ←
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]], ←
        axis=0)
    partial_train_targets = np.concatenate( ←
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]], ←
        axis=0)

    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                         validation_data=(val_data, val_targets),
                         epochs=num_epochs, batch_size=1, verbose=0)
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)

Подготовка проверочных данных:
данных из блока с номером k

```

Подготовка обучающих данных: данных из остальных блоков

Конструирование модели Keras (уже скомпилированной)

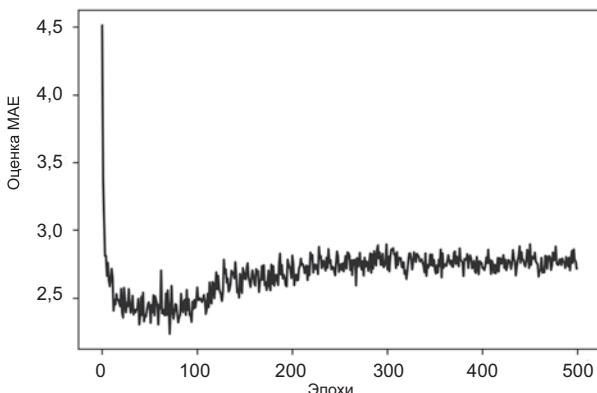
Обучение модели (в режиме без вывода сообщений, verbose = 0)

Теперь можно вычислить средние значения метрики `mae` для всех прогонов.

**Листинг 3.29.** Создание истории последовательных средних оценок проверки по К блокам

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

Построим график (рис. 3.12).



**Рис. 3.12.** Оценки MAE по эпохам

**Листинг 3.30.** Формирование графика с оценками проверок

```
import matplotlib.pyplot as plt

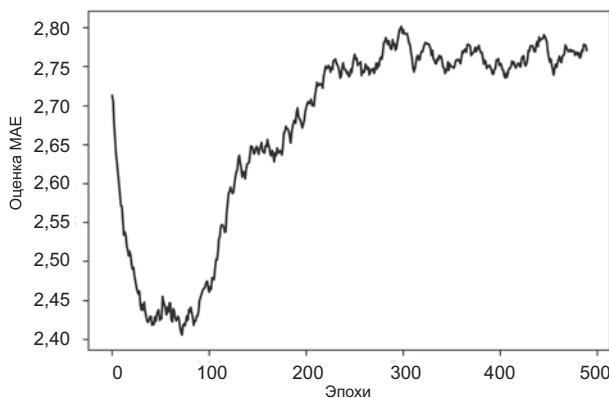
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

Из-за проблем с масштабированием, а также ввиду относительно высокой дисперсии может быть немного затруднительно увидеть общую тенденцию. Давайте сделаем следующее:

- ❑ опустим первые 10 замеров, которые имеют другой масштаб, отличный от масштаба остальной кривой;
- ❑ заменим каждую оценку экспоненциальным скользящим средним по предыдущим оценкам, чтобы получить более гладкую кривую.

Результат показан на рис. 3.13.

Согласно этому графику, наилучшая оценка MAE достигается после 80 эпох. После этого момента начинается переобучение.



**Рис. 3.13.** Оценки MAE по эпохам за исключением первых 10 замеров

**Листинг 3.31.** Формирование графика с оценками проверок за исключением первых 10 замеров

```
def smooth_curve(points, factor=0.9):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

По окончании настройки других параметров модели (кроме количества эпох можно также скорректировать количество промежуточных слоев) можно обучить окончательную версию модели на всех обучающих данных, а затем оценить ее качество на контрольных данных.

**Листинг 3.32.** Обучение окончательной версии модели

```
model = build_model()           ← Получить новую
model.fit(train_data, train_targets,   скомпилированную модель
          epochs=80, batch_size=16, verbose=0) ← Обучить ее на всем объеме
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)   обучающих данных
```

Вот окончательный результат:

```
>>> test_mae_score  
2.5532484335057877
```

Средняя ошибка все еще составляет около 2550 долларов.

### 3.6.5. Подведение итогов

Вот какие выводы вы должны сделать из этого примера:

- ❑ Регрессия выполняется с применением иных функций потерь, нежели классификация. Для регрессии часто используется функция потерь, вычисляющая среднеквадратичную ошибку (Mean Squared Error, MSE).
- ❑ Аналогично, для регрессии используются иные метрики оценки, нежели при классификации; понятие точности неприменимо для регрессии, поэтому для оценки качества часто применяется средняя абсолютная ошибка (Mean Absolute Error, MAE).
- ❑ Когда признаки образцов на входе имеют значения из разных диапазонов, их необходимо предварительно масштабировать.
- ❑ При небольшом объеме входных данных надежно оценить качество модели поможет метод перекрестной проверки по К блокам.
- ❑ При небольшом объеме обучающих данных предпочтительнее использовать маленькие сети с небольшим количеством промежуточных слоев (обычно с одним или двумя), чтобы избежать серьезного переобучения.

## Краткие итоги главы

- ❑ Теперь вы умеете решать наиболее распространенные задачи машинного обучения на векторных данных: бинарную классификацию, многоклассовую классификацию и скалярную регрессию. В разделах «Подведение итогов» выше в этой главе перечисляются наиболее важные выводы, которые вы должны извлечь из примеров решения этих задач.
- ❑ Исходные данные обычно приходится подвергать предварительной обработке перед передачей в нейронную сеть.
- ❑ Когда данные включают в себя признаки со значениями из разных диапазонов, их необходимо предварительно масштабировать.
- ❑ В процессе обучения нейронных сетей в некоторый момент наступает эффект переобучения, из-за чего падает качество результатов оценки сети на данных, которые она прежде не видела.

- ❑ При небольшом объеме входных данных используйте небольшие сети с одним или двумя промежуточными слоями, чтобы избежать серьезного переобучения.
- ❑ В том случае, когда данные делятся на большое число категорий, у вас может возникнуть узкое место для информации, если вы слишком сильно ограничите размерность промежуточных слоев.
- ❑ При регрессии применяются иные функции потерь и метрики, нежели при классификации.
- ❑ При небольшом объеме входных данных надежно оценить качество модели поможет метод перекрестной проверки по K блокам.

# 4

# Основы машинного обучения

Эта глава охватывает следующие темы:

- ✓ формы машинного обучения для решения задач помимо классификации и регрессии;
- ✓ формальные процедуры оценки моделей машинного обучения;
- ✓ подготовка данных для глубокого обучения;
- ✓ конструирование признаков;
- ✓ борьба с переобучением;
- ✓ обобщенный процесс решения задач машинного обучения.

После трех практических примеров в главе 3 у вас должно сложиться начальное понимание того, как решаются задачи классификации и регрессии с использованием нейронных сетей. Также вы собственными глазами увидели главную проблему машинного обучения — переобучение. В этой главе мы формализуем некоторые из новых знаний в прочную основу для преодоления проблем и решения задач глубокого обучения. Мы объединим все эти идеи: оценку модели, предварительную обработку данных и конструирование признаков, а также методы борьбы с переобучением — в детальный процесс решения задач машинного обучения, состоящий из семи этапов.

## 4.1. Четыре раздела машинного обучения

В предыдущих примерах вы познакомились с тремя конкретными типами задач машинного обучения: бинарной классификацией, многоклассовой классификацией и скалярной регрессией. Все три являются примерами *контролируемого обучения*, когда целью является научиться сопоставлять исходные тренировочные данные с тренировочными целями.

Контролируемое обучение — это лишь верхушка айсберга. Машинное обучение — это обширная область со сложным делением на разделы. Алгоритмы машинного обучения обычно делятся на четыре основные категории, которые описываются в следующих разделах.

### 4.1.1. Контролируемое обучение

Этот случай, безусловно, является наиболее распространенным. Его суть заключается в том, чтобы научить модель отображать исходные данные в известные целевые значения (иногда их также называют *аннотациями*) на наборе примеров (часто аннотированных людьми). Все четыре примера, которые вы видели выше в этой книге, являются каноническими примерами контролируемого обучения. Вообще говоря, к этой категории относятся почти все современные способы применения глубокого обучения, такие как распознавание образов, распознавание речи, классификация изображений и перевод с одного языка на другой.

Основную долю задач контролируемого обучения составляют классификация и регрессия, но есть и более экзотические варианты, включая следующие (с примерами):

- *Генерация последовательностей* — по заданной картинке предсказать заголовок, который ее описывает. Иногда генерацию последовательностей можно сформулировать как серию задач классификации (таких, как повторяющее предсказание следующего слова в последовательности).
- *Прогнозирование дерева синтаксиса* — по имеющемуся предложению требуется спрогнозировать его разложение в дерево синтаксиса.
- *Распознавание объектов* — на имеющейся картинке требуется нарисовать рамки вокруг определенных объектов. Эту задачу тоже можно выразить как задачу классификации (по множеству возможных рамок классифицировать содержимое каждой) или как сочетание классификации и регрессии, когда координаты рамок предсказываются посредством векторной регрессии.
- *Сегментирование изображений* — по имеющейся картинке построить пиксельную маску для конкретного объекта.

### 4.1.2. Неконтролируемое обучение

Этот раздел машинного обучения заключается в поиске интересных преобразований входных данных без помощи каких-либо целевых значений для нужд визуализации, сжатия или очистки данных от шумов или для лучшего понимания взаимосвязей в данных. Неконтролируемое обучение — это основа анализа данных, оно часто оказывается необходимым шагом на пути изучения набора данных перед применением методов контролируемого обучения. Хорошо известными примерами неконтролируемого обучения являются *понижение размерности* и *кластеризация*.

### 4.1.3. Самоконтролируемое обучение

Это разновидность контролируемого обучения, которая имеет существенные отличия, а потому может быть выделена в отдельную категорию. Самоконтролируемое обучение контролируется без использования меток, расставленных человеком, — самоконтролируемое обучение можно считать контролируемым обучением без участия людей. Здесь также имеются метки (иначе это обучение не было бы контролируемым), однако они генерируются из исходных данных, обычно с применением эвристических алгоритмов.

Хорошо известным примером самоконтролируемого обучения могут служить *автокодировщики*, которые генерируют цели по исходным немодифицированным данным. Аналогичным образом попытки предсказать следующий кадр в видео-последовательности по предыдущим кадрам или следующее слово в тексте по предыдущим словам могут служить примерами самоконтролируемого обучения (в данном случае обучения, контролируемого во времени: контроль исходит из будущих данных). Обратите внимание на то, что разница между контролируемым, самоконтролируемым и неконтролируемым обучением иногда может быть весьма условной — эти категории скорее являются областями без четких границ. Самоконтролируемое обучение можно интерпретировать как контролируемое или как неконтролируемое в зависимости от того, обращаете вы внимание на механизм обучения или на контекст его применения.

#### ПРИМЕЧАНИЕ

В этой книге мы сосредоточимся конкретно на контролируемом обучении, поскольку эта форма глубокого обучения на сегодняшний день является доминирующей и имеет широчайшую область применения. В последующих главах мы также уделим внимание самоконтролируемому обучению.

### 4.1.4. Обучение с подкреплением

Этот вид машинного обучения долгое время обходили вниманием, пока в Google DeepMind не добились успеха, обучив компьютер играть в игры Atari (а позднее в Go на высочайшем уровне). В обучении с подкреплением *агент* получает информацию о своем окружении и учится выбирать действия, максимизирующие некоторую выгоду. Например, с помощью обучения с подкреплением можно получить нейронную сеть, которая «смотрит» на экран видеоигры и руководит действиями игрока, максимизируя количество очков.

В настоящее время обучение с подкреплением пока является областью исследований и не имеет существенных практических успехов помимо применения в играх. Со временем, однако, ожидается, что обучение с подкреплением будет находить все больше и больше практических применений в реальном мире: автомобили с автопилотом, робототехника, управление ресурсами, образование и т. д. Это идея, время которой скоро наступит или только-только наступает.

## ГЛОССАРИЙ КЛАССИФИКАЦИИ И РЕГРЕССИИ

Классификация и регрессия вводят множество специальных терминов. Некоторые из них уже встречались нам в предыдущих примерах, и еще больше их встретится в следующих главах. Они имеют точные определения, характерные для машинного обучения, и вы должны знать их.

- **Образец** (*sample*), или **вход** (*input*), — один экземпляр данных, поступающий в модель.
- **Прогноз, предсказание** (*prediction*), или **выход** (*output*), — результат работы модели.
- **Цель** (*target*) — истина. То, что в идеале должна спрогнозировать модель по данным из внешнего источника.
- **Ошибка прогноза** (*prediction error*), или **значение уровня потерь** (*loss value*), — мера расстояния между прогнозом модели и целью.
- **Классы** (*classes*) — набор меток в задаче классификации, доступных для выбора. Например, в задаче классификации изображений с кошками и собаками доступны два класса: «собака» и «кошка».
- **Метка** (*label*) — конкретный экземпляр класса в задаче классификации. Например, если изображение № 1234 аннотировано как принадлежащее классу «собака», тогда «собака» является меткой для изображения № 1234.
- **Эталоны** (*ground-truth*), или **аннотации** (*annotations*), — все цели для набора данных, обычно собранные людьми.
- **Бинарная классификация** (*binary classification*) — задача классификации, которая должна разделить входные данные на две взаимоисключающие категории.
- **Многоклассовая классификация** (*multiclass classification*) — задача классификации, которая должна разделить входные данные на более чем две категории. Примером может служить классификация рукописных цифр.
- **Многозначная, или нечеткая, классификация** (*multilabel classification*) — задача классификации, в которой каждому входному образцу можно присвоить несколько меток. Например, на картинке могут быть изображены кошка и собака вместе, и поэтому такая картинка должна аннотироваться двумя метками: «кошка» и «собака». Количество меток, присваиваемых изображениям, обычно может меняться.
- **Скалярная регрессия** (*scalar regression*) — задача, в которой цель является скалярным числом, лежащим на непрерывной числовой прямой. Хорошим примером может служить прогнозирование цен на жилье: разные цены из непрерывного диапазона.
- **Векторная регрессия** (*vector regression*) — задача, в которой цель является набором чисел, лежащих на непрерывной числовой прямой. Например, регрессия по нескольким значениям (таким, как координаты прямоугольника, ограничивающего изображение) является векторной регрессией.
- **Пакет или мини-пакет** (*batch* или *mini-batch*) — небольшой набор образцов (обычно от 8 до 128), обрабатываемых моделью одновременно. Число образцов часто является степенью двойки для более эффективного использования памяти GPU. В процессе обучения один мини-пакет используется в градиентном спуске для вычисления одного изменения весов модели.

## 4.2. Оценка моделей машинного обучения

В трех примерах, представленных в главе 3, мы делили данные на тренировочный и контрольный наборы. В процессе мы быстро выяснили причину, почему нельзя оценивать качество моделей по тем же данным, на которых производилось обучение: спустя всего несколько эпох во всех трех моделях возникал эффект *переобучения*. Другими словами, качество прогнозирования на данных, которые не участвовали в обучении, прекращало возрастать (или даже ухудшалось), тогда как на тренировочных данных качество прогноза всегда растет в ходе обучения.

Цель машинного обучения состоит в создании обобщающих моделей, дающих качественный прогноз на данных, не участвовавших в обучении, а переобучение является главным препятствием к ее достижению. Вы можете контролировать только то, что наблюдаете, поэтому важно иметь возможность надежно оценивать качество обобщения вашей модели. В следующих разделах рассматриваются стратегии смягчения проблемы переобучения и достижения максимальной обобщенности. В этом разделе мы посмотрим, как измеряется обобщенность: как оценивается качество моделей машинного обучения.

### 4.2.1. Тренировочные, проверочные и контрольные наборы данных

Оценка модели всегда сводится к делению доступных данных на три набора: тренировочный, проверочный и тестовый (или контрольный). Вы обучаете модель на тренировочных данных и оцениваете с использованием проверочных. После создания окончательной версии модели вы тестируете ее с применением контрольных данных.

У вас может возникнуть вопрос: почему бы не использовать только два набора — тренировочный и контрольный? В этом случае можно было бы обучать модель на тренировочных данных и тестировать на контрольных. Ведь так намного проще!

Причина в том, что процесс конструирования модели всегда связан с настройкой ее параметров: например, с выбором количества слоев или изменением их размерности (такие настройки называют *гиперпараметрами* модели, чтобы отличать их от *параметров* — весовых коэффициентов). Для настройки в качестве сигнала обратной связи используются проверочные данные. Фактически настройка сама является разновидностью обучения: поиск более удачной конфигурации в некотором пространстве параметров. Как результат, настройка конфигурации модели по качеству прогнозирования на проверочных данных может быстро привести к *переобучению на этих данных*, даже при том, что модель никогда напрямую не обучается на них.

Главной причиной этого является так называемая *утечка информации*. Каждый раз, настраивая гиперпараметр модели и опираясь на качество прогноза по проверочным данным, вы допускаете просачивание в модель некоторой информации из этих данных. Если сделать это только один раз для одного параметра, в модель

просочится небольшой объем информации и проверочный набор данных останется надежным мерилом качества модели. Однако если повторить настройку много раз — выполняя эксперимент, оценивая модель на проверочных данных и корректируя ее по результатам — в модель будет просачиваться все больший объем информации о проверочном наборе данных.

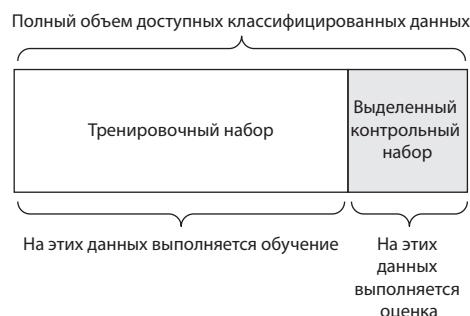
В конце концов вы получите модель, искусственно настроенную на достижение высокого качества прогнозирования по проверочным данным, потому что именно на этих данных вы ее оптимизировали. Однако истинной целью является качество прогнозирования на совершенно новых данных, не на проверочных, поэтому для оценки качества модели следует использовать отдельный набор данных, никак не участвующий в обучении, — контрольный набор. Ваша модель не должна иметь доступа *ни к какой* информации из контрольного набора, даже косвенно. Если какие-то настройки в модели выполнить на основе оценки качества прогнозирования по контрольным данным, ваша оценка обобщенности модели будет неточной.

Деление данных на тренировочный, проверочный и контрольный наборы может показаться простой задачей, тем не менее есть ряд приемов ее решения, которые могут пригодиться при ограниченном объеме исходных данных. Рассмотрим три классических рецепта оценки: проверка с простым расщеплением выборки (*hold-out validation*), перекрестная проверка по  $K$  блокам (*K-fold validation*) и итерационная проверка по  $K$  блокам с перемешиванием (*iterated K-fold validation with shuffling*).

### Проверка с простым расщеплением выборки

Некоторая часть данных выделяется в контрольный набор. Обучение производится на оставшихся данных, а оценка качества — на контрольных. Как уже говорилось в предыдущих разделах, для предотвращения утечек информации модель не должна настраиваться по результатам прогнозирования на контрольных данных, поэтому требуется *также* зарезервировать отдельный проверочный набор.

Схематически проверка с простым расщеплением выборки выглядит, как показано на рис. 4.1. В листинге 4.1 демонстрируется простейшая реализация этого приема.



**Рис. 4.1.** Деление данных при использовании проверки с простым расщеплением выборки

**Листинг 4.1.** Проверка с простым расщеплением выборки

```

num_validation_samples = 10000           | Перемешивание данных  
np.random.shuffle(data)                | нередко весьма желательно
                                             |
validation_data = data[:num_validation_samples] | Определение  
data = data[num_validation_samples:]      | проверочного набора
                                             |
training_data = data[:]                 | Определение тренировочного  
                                         | набора
                                             |
model = get_model()                    | Обучение модели на  
model.train(training_data)            | тренировочных и оценка  
validation_score = model.evaluate(validation_data) | на проверочных данных
                                             |
# В этой точке можно выполнить корректировку модели,  
# повторно обучить ее, оценить, повторить корректировку...
                                             |
model = get_model()  
model.train(np.concatenate([training_data,  
                           validation_data]))  
test_score = model.evaluate(test_data) | После настройки гиперпараметров  
                                         | часто желательно выполнить обучение  
                                         | окончательной модели на всех данных,  
                                         | не включенных в контрольный набор

```

Это самый простой протокол оценки, страдающий одним существенным недостатком: при небольшом объеме доступных данных проверочный и контрольный наборы могут содержать слишком мало образцов, чтобы считаться статистически репрезентативными. Это легко заметить: если разные случайные перестановки данных перед расщеплением дают сильно различающиеся оценки качества модели, значит, вы столкнулись именно с этой проблемой. Для ее преодоления были разработаны два других подхода: перекрестная проверка по  $K$  блокам и итерационная проверка по  $K$  блокам с перемешиванием, — которые обсуждаются ниже.

### Перекрестная проверка по $K$ блокам

При использовании этого подхода данные разбиваются на  $K$  блоков равного размера. Для каждого блока  $i$  производится обучение модели на остальных  $K-1$  блоках и оценка на блоке  $i$ . Окончательная оценка рассчитывается как среднее  $K$  промежуточных оценок. Этот метод может пригодиться, когда качество модели слишком сильно зависит от деления данных на тренировочный/контрольный наборы. Подобно проверке с простым расщеплением выборки, этот метод не избавляет от необходимости использовать отдельный проверочный набор для калибровки модели.

Схематически перекрестная проверка по  $K$  блокам выглядит, как показано на рис. 4.2. В листинге 4.2 демонстрируется простейшая реализация этого приема.

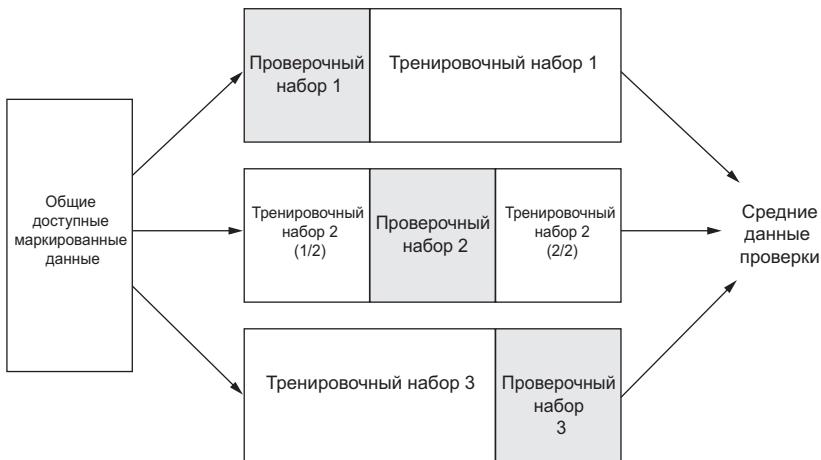


Рис. 4.2. Перекрестная проверка по трем блокам

**Листинг 4.2.** Перекрестная проверка по K блокам

```

k = 4
num_validation_samples = len(data) // k

np.random.shuffle(data)

validation_scores = []
for fold in range(k):
    validation_data = data[num_validation_samples * fold:
                           num_validation_samples * (fold + 1)]
    training_data = data[:num_validation_samples * fold] +
                    data[num_validation_samples * (fold + 1):]

    model = get_model()           ← Создание совершенно новой (необученной) модели
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)
    validation_score = np.average(validation_scores) ← Обучение окончательной модели на всех
                                                       данных, не вошедших в контрольный набор

    model = get_model()
    model.train(data)
    test_score = model.evaluate(test_data)   ← Общая оценка: среднее оценок по k блокам

```

Выбор блока данных для проверки

Использовать остальные данные для обучения. Обратите внимание: оператор + здесь выполняет конкатенацию списков, а не вычисляет сумму

**Итерационная проверка по K блокам с перемешиванием**

Этот метод подходит для ситуаций, когда имеется относительно небольшой набор данных и требуется оценить модель максимально точно. Он показал свою высокую эффективность в состязаниях на сайте Kaggle. Суть его заключается в многократном применении перекрестной проверки по  $K$  блокам с перемешиванием данных

перед каждым разделением на  $K$  блоков. Конечная оценка — среднее по оценкам, полученным в прогонах перекрестной проверки по  $K$  блокам. Обратите внимание: в конечном счете обучению и оценке подвергается  $P \times K$  моделей (где  $P$  — количество итераций), что может быть очень затратным.

### 4.2.2. Что важно помнить

Выбирая протокол оценки, всегда помните:

- *о репрезентативности данных* — наборы тренировочных и контрольных данных должны быть репрезентативными для всего объема имеющихся данных. Например, если вы пытаетесь классифицировать изображения рукописных цифр и имеете массив, в котором образцы упорядочены по классам, использование первых 80 % образцов для обучения и остальных 20 % для контроля приведет к тому, что тренировочный набор будет содержать классы 0–7, а контрольный набор — только классы 8–9. Эта ошибка может показаться смешной, однако ее совершают слишком часто. По этой причине всегда желательно *перемешивать* данные перед делением на тренировочный и контрольный наборы;
- *о направлении оси времени* — пытаясь предсказать будущее по прошлому (например, погоду на завтра, движение товаров и т. д.), вы *не* должны производить перемешивание данных перед делением, потому что это создаст *временную утечку*: ваша модель фактически будет обучаться по данным в будущем. В таких ситуациях всегда нужно следить за тем, чтобы контрольные данные *следовали непосредственно* за тренировочными данными;
- *об избыточности данных* — если некоторые образцы присутствуют в данных в нескольких экземплярах (частое явление в реальном мире), перемешивание и деление данных на тренировочный и проверочный наборы приведет к появлению избыточности между тренировочным и проверочным наборами. По сути, вы будете проводить тестирование на части тренировочных данных, что является худшим из зол! Убедитесь в том, что тренировочный и проверочный наборы не пересекаются.

## 4.3. Обработка данных, конструирование признаков и обучение признаков

Помимо оценки модели, мы должны затронуть еще один важный вопрос, прежде чем погрузиться в разработку моделей: как подготовить исходные данные и цели перед передачей их в нейронную сеть? Многие приемы обработки данных и конструирования признаков в значительной мере зависят от предметной области (например, подготовка текстовых данных или образов); мы рассмотрим их в следующих разделах на конкретных примерах. А пока познакомимся с основами, общими для всех предметных областей.

### 4.3.1. Предварительная обработка данных для нейронных сетей

Цель предварительной обработки данных — сделать исходные данные пригодными для передачи в нейронную сеть. Сюда входят векторизация, нормализация, обработка недостающих значений и извлечение признаков.

#### Векторизация

Все входы и цели в нейронной сети должны быть тензорами чисел с плавающей точкой (или, в особых случаях, тензорами целых чисел). Какие бы данные вам ни требовалось обработать — звук, изображение, текст, — их сначала нужно преобразовать в тензоры. Этот шаг называется *векторизацией данных*. Например, в двух предыдущих примерах классификации текстовых данных мы начали с того, что преобразовали текст в списки целых чисел (представляющие собой последовательности слов) и применили прямое кодирование для превращения списков в тензоры данных типа `float32`. В примерах классификации изображений цифр и предсказания цен на дома исходные данные уже имеют векторизованную форму, поэтому мы пропустили этот шаг.

#### Нормализация значений

В примере классификации цифр исходные черно-белые изображения цифр были представлены массивами целых чисел в диапазоне 0–255. Прежде чем передать эти данные в сеть, нам понадобилось привести числа к типу `float32` и разделить каждое на 255, в результате чего у нас получились массивы чисел с плавающей точкой в диапазоне 0–1. Аналогично, в примере с предсказанием цен на дома у нас имелись наборы признаков со значениями в разных диапазонах: некоторые признаки были выражены значениями с плавающей точкой, другие — целочисленными значениями. Перед передачей данных в сеть нам понадобилось нормализовать каждый признак в отдельности, чтобы все они имели среднее значение, равное 0, и стандартное отклонение, равное 1.

В общем случае небезопасно передавать в нейронную сеть данные, которые могут принимать очень большие значения (например, целые числа с большим количеством значимых разрядов, которые намного больше начальных значений, принимаемых весами сети), или разнородные данные (например, данные, в которых один признак определяется значениями в диапазоне 0–1, а другой — в диапазоне 100–200). Это может привести к значительным изменениям градиента, которые будут препятствовать сходимости сети. Чтобы упростить обучение сети, данные должны обладать следующими характеристиками:

- ❑ **принимать небольшие значения** — как правило, значения должны находиться в диапазоне 0–1;

- *быть однородными* — то есть все признаки должны принимать значения из примерно одного и того же диапазона.

Кроме того, может оказаться полезной следующая практика нормализации, хотя она не всегда необходима (например, мы не использовали нормализацию в примере классификации цифр):

- нормализация каждого признака независимо, чтобы его среднее значение было равно 0;
- нормализация каждого признака независимо, чтобы его стандартное отклонение было равно 1.

Это легко реализуется с применением массивов NumPy:

```
x -= x.mean(axis=0) ← Предполагается, что x — это двумерная матрица
x /= x.std(axis=0)  |  данных с формой (образцы, свойства)
```

## Обработка недостающих значений

Иногда в исходных данных могут отсутствовать некоторые значения. Так, в примере с предсказанием цен на дома первым признаком (столбец с индексом 0 в данных) был уровень преступности на душу населения. Как быть, если этот признак определен не во всех образцах? Если оставить все как есть, у нас будет недоставать значений в тренировочных или в контрольных данных.

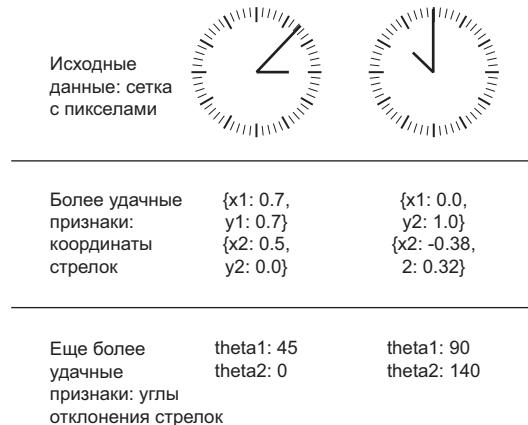
Вообще в случае с нейронными сетями вполне безопасно заменить недостающие входные значения нулями, при условии, что 0 не является осмысленным значением. Обрабатывая данные, сеть поймет, что 0 означает отсутствие данных, и будет игнорировать это значение.

Обратите внимание на то, что, если в контрольных данных имеются отсутствующие значения, но сеть была обучена без них, ваша сеть не будет распознавать отсутствующие значения! В этой ситуации следует искусственно генерировать тренировочные экземпляры с отсутствующими признаками: скопируйте несколько тренировочных образцов и отбросьте в них некоторые признаки, которые, как ожидается, не определены в контрольных данных.

### 4.3.2. Конструирование признаков

*Конструирование признаков* — это процесс использования ваших собственных знаний о данных и алгоритме машинного обучения (в данном случае — нейронной сети), чтобы улучшить эффективность алгоритма применением предопределенных преобразований к данным перед передачей их в модель. Во многих случаях не следует ожидать, что модель машинного обучения сможет обучиться на полностью произвольных данных. Данные должны передаваться в модель в виде, облегчающем ее работу.

Рассмотрим простой пример. Допустим, нам нужно разработать модель, принимающую изображение циферблата часов со стрелками и возвращающую время (рис. 4.3).



**Рис. 4.3.** Конструирование признаков для чтения времени с изображения циферблата часов

Если в качестве входных данных использовать пиксели исходных изображений, вы столкнетесь со сложной задачей машинного обучения. Для ее решения вам потребуется сконструировать сверточную сеть и потратить большой объем вычислительных ресурсов на ее обучение.

Однако, понимая задачу на высоком уровне (как человек определяет время по циферблату часов), можно сконструировать гораздо более удачные входные признаки для алгоритма машинного обучения: например, можно с легкостью написать пять строк кода на Python, которые проследуют по черным пикセルам стрелок и вернут координаты ( $x, y$ ) конца каждой стрелки. Тогда простой алгоритм машинного обучения сможет обучиться связывать эти координаты с соответствующим временем дня.

Можно пойти еще дальше: преобразовать координаты ( $x, y$ ) в полярные координаты относительно центра изображения. В этом случае на вход будут подаваться углы отклонения стрелок. Полученные в результате признаки делают задачу настолько простой, что для ее решения не требуется применять методику машинного обучения; простой операции округления и поиска в словаре вполне достаточно, чтобы получить приближенное значение времени дня.

В этом суть конструирования признаков: упростить задачу и сделать возможным ее решение более простыми средствами. Обычно для этого требуется глубокое понимание задачи.

До глубокого обучения конструирование признаков играло важную роль, поскольку классические поверхностные алгоритмы не имели пространств гипотез,

достаточно богатых, чтобы выявить полезные признаки самостоятельно. Форма данных, передаваемых алгоритму, имела решающее значение для успеха. Например, до того как нейронные сети достигли успеха, решения задачи классификации цифр из набора MNIST обычно основывались на конкретных признаках, таких как количество петель в изображении цифры, высота каждой цифры на изображении, гистограмма значений пикселов и т. д.

К счастью, современные технологии глубокого обучения в большинстве случаев избавляют от необходимости конструировать признаки, потому что нейронные сети способны автоматически извлекать полезные признаки из исходных данных. Означает ли это, что вы не должны беспокоиться о конструировании признаков при использовании глубоких нейронных сетей? Нет, и вот почему:

- ❑ Хорошие признаки позволяют решать задачи более элегантно и с меньшими затратами ресурсов. Например, было бы смешно решать проблему чтения показаний с циферблата часов с привлечением сверточной нейронной сети.
- ❑ Хорошие признаки позволяют решать задачи, имея намного меньший объем исходных данных. Способность моделей глубокого обучения самостоятельно выделять признаки зависит от наличия большого объема исходных данных; если у вас всего несколько образцов, информационная ценность их признаков приобретает определяющее значение.

## 4.4. Переобучение и недообучение

Во всех трех примерах из предыдущей главы: определение эмоциональной окраски отзыва, классификация по темам и регрессия цен на дома — качество модели на проверочных данных всегда достигало максимума после небольшого количества эпох и затем начинало падать — модель быстро достигала эффекта *переобучения*. Переобучение наблюдается во всех задачах машинного обучения. Умение справляться с этим эффектом играет важную роль в машинном обучении.

Основной проблемой машинного обучения является противоречие между оптимизацией и общностью. Под *оптимизацией* понимается процесс настройки модели для получения максимального качества на тренировочных данных (*обучение в машинном обучении*), а под *общностью* — качество обученной модели на данных, которые она прежде не видела. Цель игры — добиться высокого уровня общности, но вы не можете управлять общностью, вы можете только настраивать модель, опираясь на тренировочные данные.

В начале обучения оптимизация и общность коррелируются: чем ниже потери на тренировочных данных, тем они ниже на контрольных данных. Пока это имеет место, говорят, что модель *недообучена*: прогресс еще возможен, сеть еще не смоделировала все релевантные шаблоны в тренировочных данных. Однако после нескольких итераций на тренировочных данных общность перестает улучшаться, проверочные метрики останавливают свой рост и затем начинают ухудшаться — наступает эффект

*переобучения* модели. Другими словами, модель начинает обучаться шаблонам, характерным для тренировочных данных, но нехарактерным для новых данных.

*Лучший способ* предотвратить изучение моделью специфических или нерелевантных шаблонов, имеющих место в тренировочных данных, — *увеличить объем тренировочных данных*. Модель, обученная на большем объеме данных, будет иметь большую общность. Если это невозможно, следующим лучшим способом является регулирование качества информации или добавление ограничений на информацию, которую модели будет позволено сохранить. Если сеть может позволить себе сохранить только небольшое количество шаблонов, процесс оптимизации заставит ее сосредоточиться на наиболее существенных из них, что увеличит шансы на достижение более высокого уровня общности.

Борьба с переобучением таким способом называется *регуляризацией*. Рассмотрим некоторые распространенные приемы регуляризации и применим их на практике для улучшения модели классификации отзывов к фильмам из раздела 3.4.

#### 4.4.1. Уменьшение размера сети

Самый простой способ предотвратить переобучение — уменьшить размер модели: количество изучаемых параметров в модели (определяется количеством слоев и количеством нейронов (размерностью) в каждом слое). В глубоком обучении количество изучаемых параметров в модели часто называют *емкостью модели* (model capacity). Очевидно, что чем большим количеством параметров обладает модель, тем большим объемом памяти она обладает и, соответственно, тем легче скатиться до прямого отображения тренировочных экземпляров в их цели — отображения без обобщения. Например, модель с 500 000 бинарными параметрами легко могла бы запомнить класс каждой цифры в обучающем наборе MNIST: нам понадобится всего 10 бинарных параметров для каждой из 50 000 цифр. Но такая модель оказалась бы практически бесполезной для классификации новых образцов цифр. Всегда помните об этом: модели глубокого обучения, как правило, хорошо подогнаны под тренировочные данные, но главная задача — достижение общности, а не подгонка под данные.

С другой стороны, если сеть имеет ограниченный объем ресурсов для запоминания, она не сможет получить прямое отображение; поэтому, чтобы минимизировать потери, ей придется прибегнуть к изучению сжатых представлений, обладающих более широкими возможностями прогноза в отношении целей, — именно эти представления больше всего интересуют нас. В то же время модель должна иметь достаточное количество параметров, чтобы не возник эффект недообучения: она не должна испытывать недостатка в ресурсах для запоминания. Важно найти компромисс между *слишком большой и недостаточной емкостью*.

К сожалению, нет волшебной формулы для определения правильного количества слоев или правильного размера каждого слоя. Вам придется оценить массив разных архитектур (конечно же, опираясь на проверочный набор, но не на контрольный), чтобы определить правильный размер модели для ваших данных. В общем случае

процесс поиска подходящего размера модели должен начинаться с относительно небольшого количества слоев и параметров, а затем размеры слоев и их количество должны постепенно увеличиваться, пока не произойдет увеличение потерь на проверочных данных.

Давайте опробуем этот подход на сети, выполняющей классификацию отзывов к фильмам. В листинге 4.3 представлена исходная сеть.

#### Листинг 4.3. Исходная модель

```
from keras import models
from keras import layers

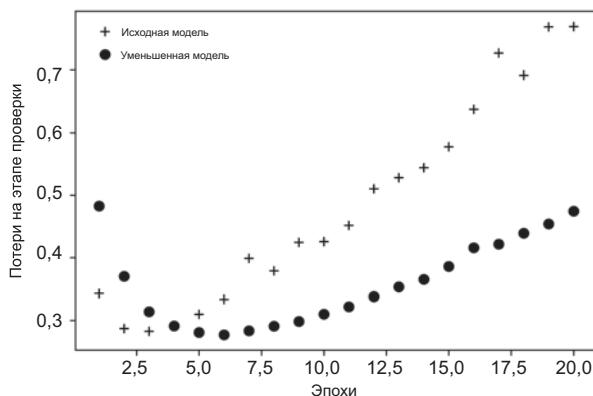
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Теперь попробуем заменить ее меньшей сетью (листинг 4.4).

#### Листинг 4.4. Версия модели с меньшей емкостью

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

На рис. 4.4 представлены для сравнения графики потерь на проверочных данных для исходной сети и уменьшенной ее версии. Точками представлены значения потерь для меньшей сети, а крестиками — для исходной (напомню, что чем ниже потери, тем выше качество модели).



**Рис. 4.4.** Влияние емкости модели на величину потерь на проверочных данных: попытка уменьшить модель

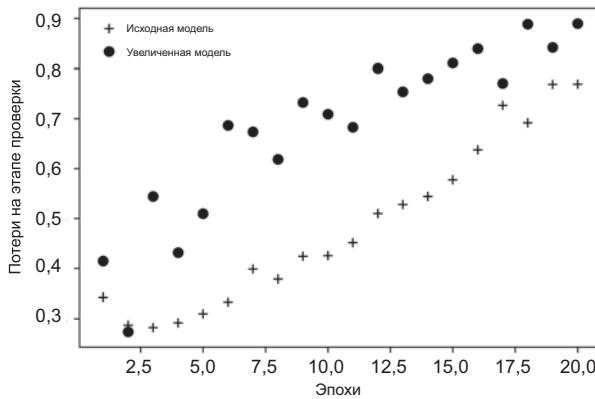
Как видите, эффект переобучения уменьшенной сети возникает позже, чем исходной (после шести эпох, а не четырех), и после переобучения ее качество ухудшается более плавно.

Теперь для контраста добавим сеть с большей емкостью — намного большей, чем необходимо для данной задачи (листинг 4.5).

#### **Листинг 4.5.** Версия модели с намного большей емкостью

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

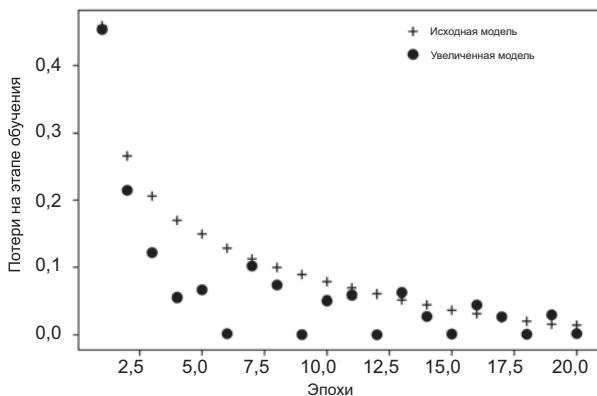
На рис. 4.5 приводятся для сравнения графики потерь на проверочных данных для большой и исходной сетей. Точками представлены значения потерь для большей сети, а крестиками — для исходной.



**Рис. 4.5.** Влияние емкости модели на величину потерь на проверочных данных: попытка увеличить модель

Переобучение увеличенной модели наступает почти сразу же, после одной эпохи, и имеет намного более серьезные последствия. Результаты измерения потерь на этапе проверки оказываются слишком искаженными.

На рис. 4.6 приводятся графики потерь на тренировочных данных для этих двух сетей. Как видите, большая сеть быстро достигает нулевых потерь на тренировочных данных. Чем больше емкость сети, тем быстрее она обучается моделировать тренировочные данные (что приводит к очень низким потерям на тренировочных данных), но она более восприимчива к переобучению (приводящему к большой разности между потерями на тренировочных и проверочных данных).



**Рис. 4.6.** Влияние емкости модели на величину потерь на тренировочных данных: попытка увеличить модель

#### 4.4.2. Добавление регуляризации весов

Возможно вы знакомы с принципом *бритвы Оккама*: если какому-то явлению можно дать два объяснения, правильным, скорее всего, будет более простое — имеющее меньшее количество допущений. Эта идея применима также к моделям, полученным с помощью нейронных сетей: для одних и тех же наборов тренировочных данных и архитектуры сети существует множество наборов весовых значений (*моделей*), объясняющих данные. Более простые модели менее склонны к переобучению, чем сложные.

*Простая модель* в данном контексте — это модель, в которой распределение значений параметров имеет меньшую энтропию (или модель с меньшим числом параметров, как было показано в предыдущем разделе). То есть типичный способ смягчения проблемы переобучения заключается в уменьшении сложности сети путем ограничения значений ее весовых коэффициентов, что делает их распределение более *равномерным*. Этот прием называется *регуляризацией весов*, он реализуется добавлением в функцию потерь сети *штрафа за увеличение весов* и имеет две разновидности:

- *L1-регуляризация* (L1 regularization) — добавляемый штраф прямо пропорционален *абсолютным значениям весовых коэффициентов* (L1-норма весов).
- *L2-регуляризация* (L2 regularization) — добавляемый штраф пропорционален *квадратам значений весовых коэффициентов* (L2-норма весов). В контексте нейронных сетей L2-регуляризация также называется *сокращением весов* (weight decay). Это два разных названия одного и того же явления: сокращение весов с математической точки зрения суть то же самое, что L2-регуляризация.

В Keras регуляризация весов осуществляется путем передачи в слои именованных аргументов с *экземплярами регуляризаторов весов*. Рассмотрим пример добавления L2-регуляризации в сеть классификации отзывов о фильмах (листинг 4.6).

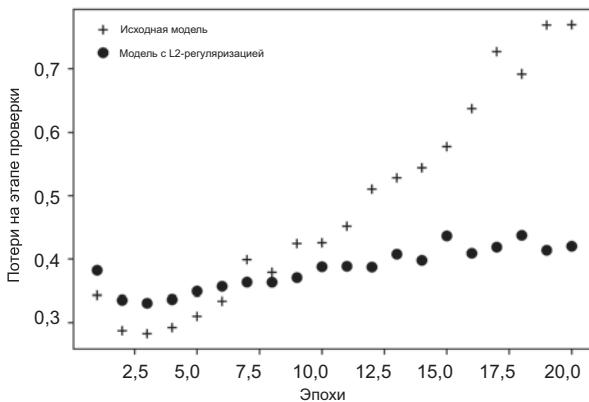
**Листинг 4.6.** Добавление L2-регуляризации весов в модель

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                      activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                      activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

`l2(0.001)` означает, что каждый коэффициент в матрице весов слоя будет добавлять  $0.001 * \text{weight\_coefficient\_value}$  в общее значение потерь сети. Обратите внимание: так как *штраф добавляется только на этапе обучения*, величина потерь сети на этапе обучения будет намного выше, чем на этапе контроля.

На рис. 4.7 показано влияние L2-регуляризации. Как видите, модель с L2-регуляризацией (точки) намного устойчивее к переобучению, чем исходная модель (крестики), даже при том, что обе модели имеют одинаковое количество параметров.



**Рис. 4.7.** Влияние L2-регуляризации весов на величину потерь на проверочных данных

Вместо L2-регуляризации можно также использовать следующие регуляризаторы, входящие в состав Keras.

**Листинг 4.7.** Разные регуляризаторы, доступные в Keras

```
from keras import regularizers
```

```
regularizers.l1(0.001) ← L1-регуляризация
```

```
regularizers.l1_l2(l1=0.001, l2=0.001) ← Объединенная L1- и L2-регуляризация
```

### 4.4.3. Добавление прореживания

*Прореживание* (dropout) — один из наиболее эффективных и распространенных приемов регуляризации для нейронных сетей, разработанный Джоном Хинтоном (Geoff Hinton) и его студентами в Университете Торонто. Прореживание, которое применяется к слою, заключается в *удалении* (присваивании нуля) случайно выбираемым признакам на этапе обучения. Представьте, что в процессе обучения некоторый уровень для данного образца на входе в нормальной ситуации возвращает вектор  $[0.2, 0.5, 1.3, 0.8, 1.1]$ . После применения прореживания некоторые элементы вектора получают нулевое значение: например,  $[0, 0.5, 1.3, 0, 1.1]$ . *Коэффициент прореживания* — это доля обнуляемых признаков; обычно он выбирается в диапазоне от 0,2 до 0,5. На этапе тестирования прореживание не производится; вместо этого выходные значения уровня уменьшаются на коэффициент, равный коэффициенту прореживания, чтобы компенсировать разницу в активности признаков на этапах тестирования и обучения.

Рассмотрим матрицу NumPy, содержащую результат слоя, `layer_output`, с формой (размер\_пакета, признаки). На этапе обучения мы обнуляем случайно выбираемые значения в матрице:

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape) ←  
На этапе обучения обнуляется  
50% признаков в выводе
```

На этапе тестирования мы уменьшаем результаты на коэффициент прореживания. В данном случае на коэффициент 0,5 (потому что прежде была отброшена половина признаков):

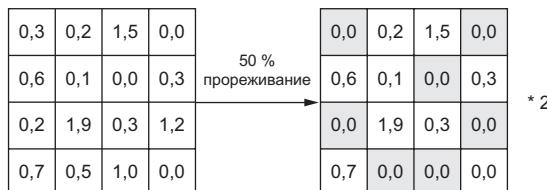
```
layer_output *= 0.5 ← На этапе тестирования
```

Обратите внимание на то, что этот процесс можно реализовать полностью на этапе обучения и оставить без изменения результаты, получаемые на этапе тестирования, что часто и делается на практике (рис. 4.8):

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape) ←  
layer_output /= 0.5 ←  
Обратите внимание: в данном  
случае происходит увеличение,  
а не уменьшение значений
```

На этапе обучения

Этот прием может показаться странным и необоснованным. Каким образом он поможет справиться с переобучением? По словам Хинтона, основой для этого приема, кроме всего прочего, стал механизм, используемый банками для предотвращения мошенничества. Вот его слова: «Посещая свой банк, я заметил, что операционисты, обслуживающие меня, часто меняются. Я спросил одного из них, почему так происходит. Он сказал, что не знает, но им часто приходится переходить с места на место. Я предположил, что это делается для исключения мошеннического говора клиента с сотрудником банка. Это навело меня на мысль, что удаление случайно



**Рис. 4.8.** Прореживание применяется к матрице активации на этапе обучения с масштабированием на этом же этапе. На этапе тестирования матрица активации не изменяется

выбранного подмножества нейронов из каждого примера может помочь предотвратить заговор модели с исходными данными и тем самым ослабить эффект переобучения<sup>1</sup>. Основная идея заключается в том, что введение шума в выходные значения может разбить случайно складывающиеся шаблоны, не имеющие большого значения (Хинтон называет их *заговорами*), которые модель начинает запоминать в отсутствие шума.

В Keras добавить прореживание в сеть можно посредством уровня `Dropout`, который обрабатывает результаты работы слоя, стоящего непосредственно перед ним:

```
model.add(layers.Dropout(0.5))
```

Давайте добавим два слоя `Dropout` в сеть IMDB и посмотрим, как это повлияет на эффект переобучения.

#### Листинг 4.8. Добавление прореживания в сеть IMDB

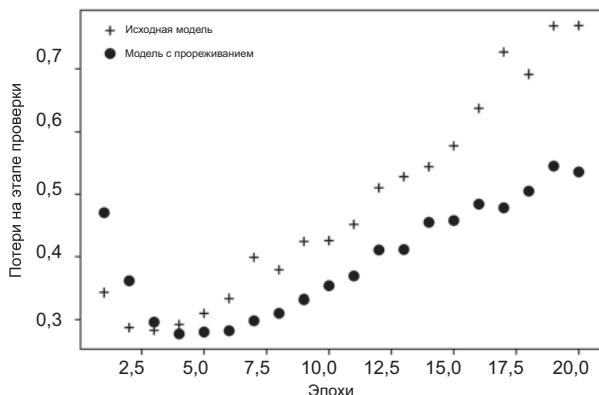
```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

На рис. 4.9 показаны графики с результатами. И снова наблюдается улучшение в сравнение с оригинальной сетью.

В качестве напоминания ознакомьтесь с наиболее распространенными способами ослабления проблемы переобучения нейронных сетей:

- увеличить объем тренировочных данных;
- уменьшить емкость сети;
- добавить регуляризацию весов;
- добавить прореживание.

<sup>1</sup> См. ветку обсуждения на сайте Reddit: «AMA: We are the Google Brain team. We'd love to answer your questions about machine learning» (AMA: мы — команда из Google Brain. Мы с удовольствием ответим на ваши вопросы о машинном обучении), <http://mng.bz/XrsS>.



**Рис. 4.9.** Влияние прореживания на величину потерь на проверочных данных

## 4.5. Обобщенный процесс решения задач машинного обучения

Мы представим универсальную схему, которую вы сможете использовать для решения любых задач машинного обучения. Эта схема связывает воедино идеи, описанные в этой главе: определение задачи, оценка, конструирование признаков и ослабление проблемы переобучения.

### 4.5.1. Определение задачи и создание набора данных

Прежде всего, необходимо определить задачу:

- ❑ Какой вид будут иметь входные данные? Что требуется предсказать? Вы сможете обучить сеть предсказывать что-либо только при наличии тренировочных данных: например, обучить сеть определять эмоциональную окраску отзывов к фильмам можно, если имеются отзывы и соответствующие аннотации. То есть доступность данных на этом этапе является ограничивающим фактором (у вас может не быть средств, чтобы заплатить людям, которые соберут для вас данные).
- ❑ К какому типу относится задача, стоящая перед вами? Бинарная классификация? Многоклассовая классификация? Скалярная регрессия? Векторная регрессия? Многоклассовая, многозначная (нечеткая) классификация? Чего иное, например кластеризация, генерация или обучение с подкреплением? Идентификация типа задачи определит выбор архитектуры модели, функции потерь и т. д.

Вы не сможете перейти к следующему этапу, не зная, какие данные будут подаваться на вход и что должно получиться на выходе. Помните о гипотезах, которые вы ставите на этой стадии:

- гипотеза о том, что выходные данные можно предсказать по входным данным;
- гипотеза о том, что доступные данные достаточно информативны для изучения отношений между входными и выходными данными.

Пока у вас нет рабочей модели, это всего лишь гипотезы, ожидающие подтверждения или опровержения. Не все задачи имеют решение; наличие входных данных  $X$  и целей  $Y$  еще не означает, что  $X$  содержит достаточно информации для предсказания  $Y$ . Например, если вы пытаетесь предсказать движение акций на фондовой бирже по недавней истории изменения цен, вы едва ли добьетесь успеха, потому что история цен не содержит достаточного объема информации для уверенного прогнозирования.

Один из классов неразрешимых задач, о которых вы должны знать, — нестационарные задачи. Представьте, что вы пытаетесь создать механизм рекомендаций: вы обучили модель на данных, собранных за один месяц (август), и теперь хотите начать генерировать рекомендации на зимние месяцы. Самая большая проблема в том, что в разные сезоны люди покупают разную одежду: покупка одежды — не стационарное явление в масштабе нескольких месяцев. То, что вы пытаетесь моделировать, меняется с течением времени. В данном случае правильным шагом будет постоянное переучивание модели на данных, собранных в недавнем прошлом, или сбор данных в масштабе времени, когда задача остается стационарной. Для циклически меняющихся задач, таких как покупка одежды, нужно собрать данные за несколько лет, чтобы смоделировать сезонные изменения, — но не забудьте сделать время года одним из входных параметров вашей модели!

Имейте в виду, что машинное обучение можно использовать только для запоминания шаблонов, присутствующих в тренировочных данных. Модель сможет распознавать только то, что видела прежде. Использование моделей, обученных по данным из прошлого, для предсказания будущего возможно, только если будущее будет вести себя так же, как прошлое. Что в действительности встречается очень редко.

## 4.5.2. Выбор меры успеха

Чтобы держать ситуацию под контролем, нужно иметь возможность наблюдать за ней. Чтобы добиться успеха, важно определить, что понимается под успехом — близость? Точность и полнота? Удержание клиентов? Ваш показатель успеха будет определять выбор функции потерь — что именно должна оптимизировать ваша модель. Он должен быть прямо связан с вашими общими целями, такими как успех бизнеса, например.

Для задач симметричной классификации, когда каждый класс одинаково вероятен, часто используются такие показатели, как близость и *площадь под кривой рабочей характеристики приемника* (Area Under Curve of Receiver Operating Characteristic, ROC AUC). Для задач несимметричной классификации можно использовать точность и полноту. Для задач ранжирования или многозначной классификации можно использовать среднее математическое ожидание. Также нередко приходится определять собственную меру успеха. Чтобы получить представление о разнообразии мер успеха в машинном обучении и их связях с разными предметными областями, полезно ознакомиться с состязаниями аналитиков на сайте Kaggle (<https://kaggle.com>); там вы увидите широкий спектр проблем и оцениваемых показателей.

### 4.5.3. Выбор протокола оценки

После определения цели необходимо также выяснить, как измерять движение к ней. Выше мы рассмотрели три распространенных протокола оценки:

- ❑ *выделение из общей выборки отдельного проверочного набора данных* — этот способ хорошо подходит при наличии большого объема данных;
- ❑ *перекрестная проверка по K блокам* — правильный выбор при небольшом количестве исходных образцов, из которых нельзя выделить представительную выборку для проверки;
- ❑ *итерационная проверка по K блокам с перемешиванием* — позволяет с высокой точностью оценить модель, когда в вашем распоряжении имеется ограниченный объем данных.

Просто выберите один из этих. В большинстве случаев первый поможет получить достаточно надежную оценку.

### 4.5.4. Предварительная подготовка данных

После того как определена задача и исходные данные для обучения, цель для оптимизации и порядок оценки предпринятого подхода, у вас есть практически все, чтобы начать обучение модели. Но прежде необходимо преобразовать исходные данные в формат, в котором их можно передать в модель машинного обучения — в данном случае в глубокую нейронную сеть:

- ❑ как было показано выше, данные должны быть помещены в тензоры;
- ❑ значения, помещаемые в тензоры, обычно требуют масштабирования и приведения к меньшим величинам: например, в диапазоне  $[-1, 1]$  или  $[0, 1]$ ;
- ❑ если значения разных признаков находятся в разных диапазонах (разнородные данные), их следует нормализовать;
- ❑ возможно, вам также понадобится выполнить конструирование признаков, особенно при небольшом объеме исходных данных.

После того как тензоры с исходными и целевыми данными будут готовы, можно приступать к обучению модели.

#### 4.5.5. Разработка модели, более совершенной, чем базовый случай

Ваша цель на этом этапе — достичь статистической мощности, то есть разработать небольшую модель, способную выдать более качественный результат по сравнению с базовым случаем. В примере классификации рукописных цифр из набора MNIST про любую модель, достигшую точности выше 0,1, можно сказать, что она обладает статистической мощностью; в примере IMDB таковой можно назвать любую модель с точностью выше 0,5.

Обратите внимание на то, что не всегда удается достичь статистической мощности. Если модель не в состоянии дать более высокую точность, чем простой случайный выбор (базовый случай) после опробования нескольких разумных архитектур, вполне возможно, что во входных данных отсутствует ответ на вопрос, который вы пытаетесь задать. Не забывайте, что вы ставите две гипотезы:

- гипотеза о том, что выходные данные можно предсказать по входным данным;
- гипотеза о том, что доступные данные достаточно информативны для изучения отношений между входными и выходными данными.

Вполне возможно, что эти гипотезы ложны, и тогда вам придется выполнить проектирования с самого начала.

Если все идет как надо, вам нужно сделать три ключевых выбора для создания первой рабочей модели:

- Функция активации для последнего уровня* — устанавливает эффективные ограничения на результат сети. Например, в случае классификации отзывов из IMDB на последнем уровне используется функция `sigmoid`, в случае регрессии вообще не используется функция активации на последнем уровне и т. д.
- Функция потерь* — должна соответствовать типу решаемой задачи. Например, в случае IMDB используется функция потерь `binary_crossentropy`, в случае регрессии используется функция `mse` и т. д.
- Конфигурация оптимизации* — какой оптимизатор использовать? Какой выбрать шаг обучения? В большинстве случаев с успехом можно использовать `rmsprop` с шагом обучения по умолчанию.

Выбирая функцию потерь, имейте в виду, что не всегда можно напрямую оптимизировать показатель успеха решения задачи. Иногда нет простого способа преобразовать показатель успеха в функцию потерь; функции потерь, в конце концов, должны быть вычислимые на мини-пакетах данных (в идеале функция потерь должна быть вычислимой на очень маленьких объемах данных, вплоть до одного

экземпляра) и дифференцируемыми (иначе не получится использовать обратное распространение ошибки для обучения сети). Например, широко используемую метрику классификации ROC AUC нельзя оптимизировать непосредственно. Поэтому в задачах классификации обычно оптимизируется некоторая ее оценка, например перекрестная энтропия. В общем случае можно считать, что чем ниже величина перекрестной энтропии, тем выше будет значение ROC AUC.

Таблица 4.1 поможет вам выбрать функцию активации для последнего уровня и функцию потерь для некоторых типичных задач.

**Таблица 4.1.** Выбор функции активации для последнего уровня и функции потерь

Тип задачи	Функция активации для последнего уровня	Функция потерь
Бинарная классификация	sigmoid	binary_crossentropy
Многоклассовая, однозначная классификация	softmax	categorical_crossentropy
Многоклассовая, многозначная классификация	sigmoid	binary_crossentropy
Регрессия по произвольным значениям	Нет	mse
Регрессия по значениям между 0 и 1	sigmoid	mse или binary_crossentropy

#### 4.5.6. Масштабирование по вертикали: разработка модели с переобучением

После получения модели, обладающей статистической мощностью, встает вопрос о достаточной мощности модели. Достаточно ли слоев и параметров, чтобы правильно смоделировать задачу? Например, сеть с единственным скрытым (промежуточным) слоем, имеющим два параметра, будет иметь некоторую статистическую мощность для классификации цифр из набора MNIST, но недостаточную, чтобы считать задачу решенной. Не забывайте о распространенной проблеме машинного обучения — противоречии между оптимизацией и общностью; идеальной считается модель, которая стоит непосредственно на границе между недообучением и переобучением, между недостаточной и избыточной емкостью. Чтобы понять, где проходит эта граница, ее сначала нужно пересечь.

Чтобы понять, насколько большой должна быть модель, сначала нужно сконструировать модель, обладающую эффектом переобучения. Сделать это просто:

1. Добавьте слои.
2. Задайте большое количество параметров в слоях.
3. Обучите модель на большом количестве эпох.

Постоянно контролируйте, как меняется уровень потерь на этапах обучения и проверки, а также любые другие показатели на этих же этапах, которые вас интересуют. Ухудшение качества модели на проверочных данных свидетельствует о достижении эффекта переобучения.

Следующий этап — регуляризация и настройка модели, чтобы как можно ближе подойти к идеальной модели, которая не страдает ни недообучением, ни переобучением.

#### 4.5.7. Регуляризация модели и настройка гиперпараметров

Этот шаг занимает больше всего времени: вам придется многократно изменять свою модель, обучать ее, оценивать качество на проверочных данных (контрольные данные не должны принимать никакого участия в этом этапе), снова изменять ее и повторять этот цикл, пока качество модели не достигнет желаемого уровня. Вот кое-что из того, что вы должны попробовать:

- ❑ добавить прореживание;
- ❑ опробовать разные архитектуры: добавлять и удалять слои;
- ❑ добавить L1- и (или) L2-регуляризацию;
- ❑ опробовать разные гиперпараметры (например, число нейронов на слой или шаг обучения оптимизатора), чтобы найти оптимальные настройки;
- ❑ дополнительно можно выполнить цикл конструирования признаков: добавить новые признаки или удалить имеющиеся, которые не кажутся информативными.

Помните: каждый раз, когда вы используете обратную связь из процесса проверки для настройки модели, происходит утечка информации в модель. Если цикл повторяется лишь несколько раз, в этом нет ничего страшного; но если цикл проверки и настройки выполняется многократно, в конечном итоге это приведет к переобучению модели на проверочных данных (даже при том, что модель напрямую не получает их). Это снижает надежность процесса оценки.

Получив удовлетворительную конфигурацию, можно обучить окончательный вариант модели на всех доступных данных (тренировочных и проверочных) и оценить ее качество на контрольном наборе. Если качество модели на контрольных данных окажется значительно хуже, чем на проверочных, это может означать, что ваша процедура проверки была ненадежной или в процессе настройки параметров модели проявился эффект переобучения на проверочных данных. В этом случае можно попробовать переключиться на использование другого, более надежного протокола оценки (такого, как итерационная проверка по К блокам с перемешиванием).

## Краткие итоги главы

- Определите задачу и данные, на которых будет проводиться обучение. Соберите эти данные и, если нужно, аннотируйте их.
- Выберите меру успеха — показатели, которые можно было бы отслеживать по проверочным данным.
- Выберите протокол оценки: будете ли вы оценивать по выделенному из общей выборки проверочному набору данных или методом перекрестной проверки по К блокам? Какую часть данных вы могли бы использовать для проверки?
- Разработайте первую модель, более совершенную, чем базовый случай: модель, обладающую статистической мощностью.
- Разработайте модель, страдающую эффектом переобучения.
- Выполните регуляризацию модели и настройте ее гиперпараметры, опираясь на оценку качества по проверочным данным. Многие исследования в области машинного обучения сосредоточены исключительно на этом шаге — однако не упускайте из виду общую картину.

## ЧАСТЬ II

# Глубокое обучение на практике

Главы 5–9 помогут вам получить представление о способах решения практических задач с применением глубокого обучения. Большинство примеров программного кода сконцентрировано в этой части.

# 5

# Глубокое обучение в технологиях компьютерного зрения

Эта глава охватывает следующие темы:

- ✓ суть сверточных нейронных сетей;
- ✓ расширение обучающего набора данных для ослабления эффекта переобучения;
- ✓ использование предварительно обученной сверточной нейронной сети для извлечения признаков;
- ✓ дообучение предварительно обученной сверточной нейронной сети;
- ✓ визуализация процессов обучения сверточных нейронных сетей и принятия ими решений.

Эта глава знакомит со сверточными нейронными сетями — разновидностью моделей глубокого обучения, почти повсеместно используемых в приложениях компьютерного зрения (распознавания образов). Здесь вы научитесь применять сверточные нейронные сети для решения задач классификации изображений, в частности задач с небольшими наборами обучающих данных, которые являются наиболее распространенными, если только вы не работаете в крупной технологической компании.

## 5.1. Введение в сверточные нейронные сети

В этом разделе мы погрузимся в теорию сверточных нейронных сетей и выясним причины их успеха в задачах распознавания образов. Но сначала рассмотрим практический пример простой сверточной нейронной сети, в котором сеть используется для классификации изображений цифр из набора MNIST. Эту задачу мы решили

в главе 2, использовав полносвязную сеть (ее точность на контрольных данных составила 97,8 %). Несмотря на простоту сверточной нейронной сети, ее точность будет значительно выше полносвязной модели из главы 2.

В листинге 5.1 показано, как выглядит простая сверточная нейронная сеть. Это стек слоев Conv2D и MaxPooling2D. Как она действует, рассказывается чуть ниже.

### **Листинг 5.1.** Создание небольшой сверточной нейронной сети

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Важно отметить, что данная сеть принимает на входе тензоры с формой (**высота\_изображения, ширина\_изображения, каналы**) (не включая измерение, определяющее пакеты). В данном случае мы настроили сеть на обработку входов с размерами (28, 28, 1), соответствующими формату изображений в наборе MNIST, передав аргумент `input_shape=(28, 28, 1)` в первый слой.

Рассмотрим поближе текущую архитектуру сети:

```
>>> model.summary()

Layer (type)                 Output Shape              Param #
================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)      320
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)      0
conv2d_2 (Conv2D)            (None, 11, 11, 64)      18496
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)        0
conv2d_3 (Conv2D)            (None, 3, 3, 64)        36928
================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

Как видите, все слои, Conv2D и MaxPooling2D, выводят трехмерный тензор с формой (**высота, ширина, каналы**). Измерения ширины и высоты сжимаются с ростом глубины сети. Количество каналов управляется первым аргументом, передаваемым в слои Conv2D (32 или 64).

Следующий шаг — передача последнего выходного тензора (с формой (3, 3, 64)) на вход полно связной классифицирующей сети, подобной той, с которой мы уже знакомы: стека слоев `Dense`. Эти классификаторы обрабатывают векторы — одномерные массивы, тогда как текущий выход является трехмерным тензором. Поэтому мы должны прежде преобразовать трехмерный вывод в одномерный и затем добавить сверху несколько слоев `Dense`.

**Листинг 5.2.** Добавление классификатора поверх сверточной нейронной сети

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Мы реализуем 10-видовую классификацию, используя конечный слой с 10 выходами и функцией активации `softmax`. Вот как выглядит сеть теперь:

```
>>> model.summary()
Layer (type)                  Output Shape           Param #
=====
conv2d_1 (Conv2D)             (None, 26, 26, 32)    320
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32)    0
conv2d_2 (Conv2D)             (None, 11, 11, 64)    18496
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)      0
conv2d_3 (Conv2D)             (None, 3, 3, 64)      36928
flatten_1 (Flatten)           (None, 576)            0
dense_1 (Dense)               (None, 64)             36928
dense_2 (Dense)               (None, 10)             650
=====
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

Как видите, выходы (3, 3, 64) преобразуются в векторы с формой (576,) перед передачей двум слоям `Dense`.

Теперь давайте обучим сеть, передав ей цифры из набора MNIST. Далее мы будем повторно использовать большое количество программного кода из главы 2.

**Листинг 5.3.** Обучение сверточной нейронной сети на данных из набора MNIST

```
from keras.datasets import mnist
from keras.utils import to_categorical
```

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Оценим модель на контрольных данных:

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
0.9908000000000001
```

Полносвязная сеть из главы 2 показала точность 97,8 % на контрольных данных, а простенькая сверточная нейронная сеть показала точность 99,3 %: мы уменьшили процент ошибок на 68 % (относительно). Неплохо!

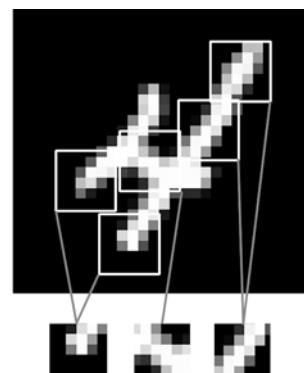
Но почему такая простая сверточная нейронная сеть работает настолько хорошо в сравнении с полносвязной моделью? Чтобы ответить на этот вопрос, погрузимся в особенности работы слоев Conv2D и MaxPooling2D.

### 5.1.1. Операция свертывания

Основное отличие полносвязного слоя от сверточного заключается в следующем: слои Dense изучают глобальные шаблоны в пространстве входных признаков (например, в случае с цифрами из набора MNIST это шаблоны, вовлекающие все пиксели), тогда как сверточные слои изучают локальные шаблоны (рис. 5.1): в случае с изображениями — шаблоны в небольших двумерных окнах во входных данных. В предыдущем примере все такие окна имели размеры  $3 \times 3$ .

Эта ключевая характеристика наделяет сверточные нейронные сети двумя важными свойствами:

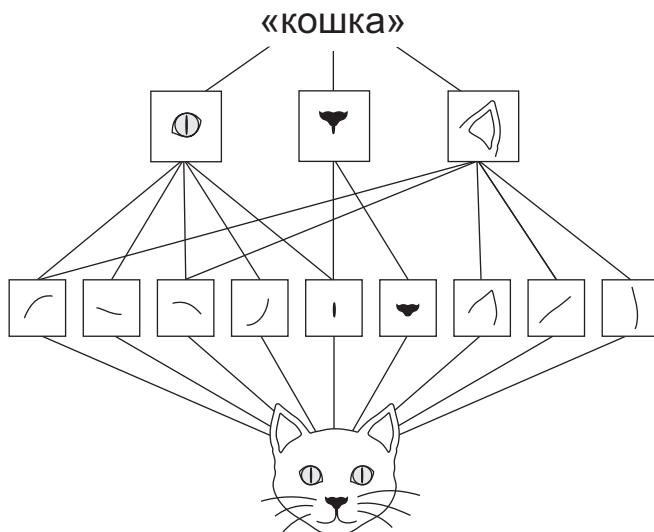
- ❑ *Шаблоны, которые они изучают, являются инвариантными в отношении переноса.* После изучения определенного шаблона в правом нижнем углу картинки сверточная нейронная сеть сможет рас-



**Рис. 5.1.** Изображения можно разбить на локальные шаблоны, такие как края, текстуры и т. д.

познавать его повсюду: например, в левом верхнем углу. Полносвязной сети пришлось бы изучить шаблон заново, если он появляется в другом месте. Это увеличивает эффективность сверточных сетей в задачах обработки изображений (потому что *видимый мир по своей сути является инвариантным в отношении переноса*): таким сетям требуется меньше обучающих образцов для получения представлений, обладающих силой обобщения.

- Они могут изучать пространственные иерархии шаблонов (рис. 5.2). Первый сверточный слой будет изучать небольшие локальные шаблоны, такие как края, второй — более крупные шаблоны, состоящие из признаков, возвращаемых первым слоем, и т. д. Это позволяет сверточным нейронным сетям эффективно изучать все более сложные и абстрактные визуальные представления (потому что *видимый мир по своей сути является пространственно-иерархическим*).

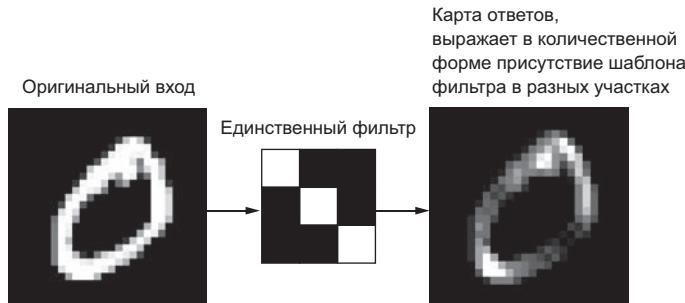


**Рис. 5.2.** Видимый мир формируется пространственными иерархиями видимых модулей: гиперлокальные края объединяются в локальные объекты, такие как глаза или уши, которые, в свою очередь, объединяются в понятия еще более высокого уровня, такие как «кошка»

Свертка применяется к трехмерным тензорам, называемым *картами признаков*, с двумя пространственными осями (высота и ширина), а также с осью глубины (или осью каналов). Для изображений в формате RGB размерность оси глубины равна 3, потому что имеется три канала цвета: красный (red), зеленый (green) и синий (blue). Для черно-белых изображений, как в наборе MNIST, ось глубины имеет размерность 1 (оттенки серого). Операция свертывания извлекает шаблоны из своей входной карты признаков и применяет одинаковые преобразования ко всем шаблонам, производя *выходную карту признаков*. Эта выходная карта признаков также является

трехмерным тензором: она имеет ширину и высоту. Ее глубина может иметь любую размерность, потому что выходная глубина является параметром слоя, и разные каналы на этой оси глубины больше не соответствуют конкретным цветам, как во входных данных в формате RGB, скорее они соответствуют *фильтрам*. Фильтры представляют собой конкретные аспекты входных данных: на верхнем уровне, например, фильтр может соответствовать понятию «присутствие лица на входе».

В примере MNIST первый сверточный слой принимает карту признаков с размером  $(28, 28, 1)$  и выводит карту признаков с размером  $(26, 26, 32)$ : он вычисляет 32 фильтра по входным данным. Каждый из этих 32 выходных каналов содержит сетку  $26 \times 26$  значений — *карту ответов* фильтра на входных данных, определяющую ответ этого шаблона фильтра для разных участков входных данных (рис. 5.3). Вот что означает термин *карта признаков*: каждое измерение на оси глубины — это признак (или фильтр), а двумерный тензор `output[:, :, n]` — это двумерная пространственная *карта ответов* этого фильтра на входных данных.



**Рис. 5.3.** Понятие карты ответов: двумерная карта присутствия шаблона в разных участках входных данных

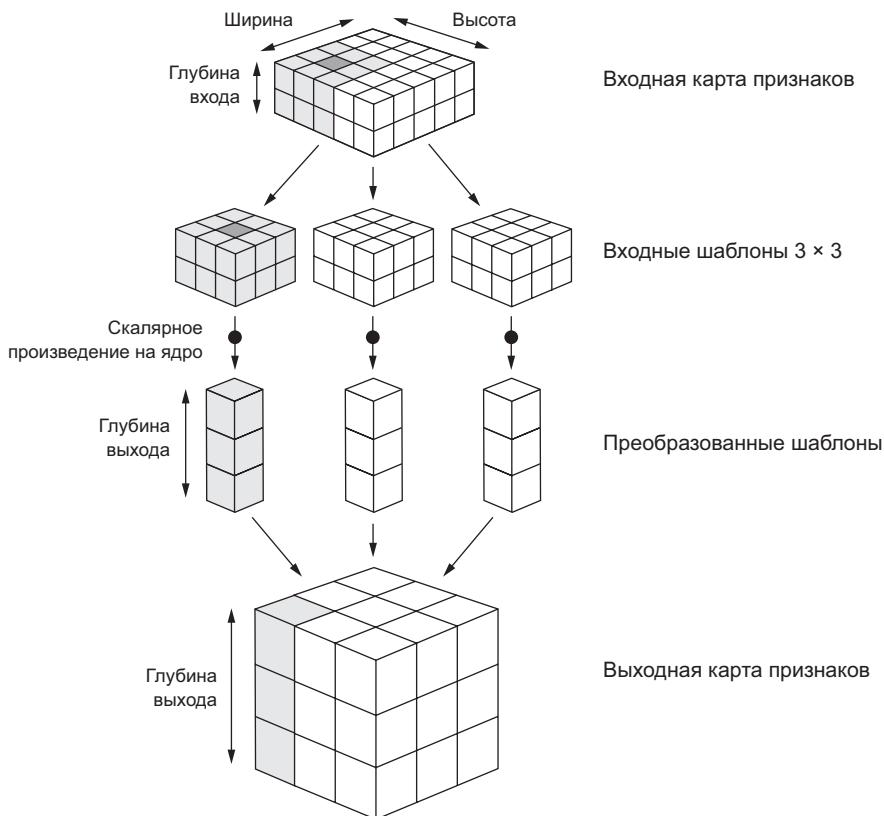
Свертки определяются двумя ключевыми параметрами:

- ❑ *Размер шаблонов, извлекаемых из входных данных*, — обычно  $3 \times 3$  или  $5 \times 5$ . В данном примере используется размер  $3 \times 3$ , что является распространенным выбором.
- ❑ *Глубина выходной карты признаков* — количество фильтров, вычисляемых сверткой. В данном примере свертка начинается с глубины 32 и заканчивается глубиной 64.

В Keras эти параметры передаются в слои `Conv2D` в первых аргументах: `Conv2D(выходная_глубина, (высота_окна, ширина_окна))`.

Свертка работает методом скользящего окна: она *двигает* окно с размером  $3 \times 3$  или  $5 \times 5$  по трехмерной входной карте признаков, останавливается в каждой возможной позиции и извлекает трехмерный шаблон окружающих признаков (с формой `(высота_окна, ширина_окна, глубина_входа)`). Каждый такой трехмерный шаблон затем преобразуется (путем умножения тензора на матрицу весов,

получаемую в ходе обучения, которая называется *ядром свертки*) в одномерный вектор с формой (*выходная глубина*,). Все эти векторы затем собираются в трехмерную выходную карту с формой (*высота*, *ширина*, *выходная глубина*). Каждое пространственное местоположение в выходной карте признаков соответствует тому же местоположению во входной карте признаков (например, правый нижний угол выхода содержит информацию о правом нижнем угле входа). Например, для окна  $3 \times 3$  вектор  $\text{output}[i, j, :]$  соответствует трехмерному шаблону  $\text{input}[i-1:i+2, j-1:j+2, :]$ . Полный процесс изображен на рис. 5.4.



**Рис. 5.4.** Принцип действия свертки

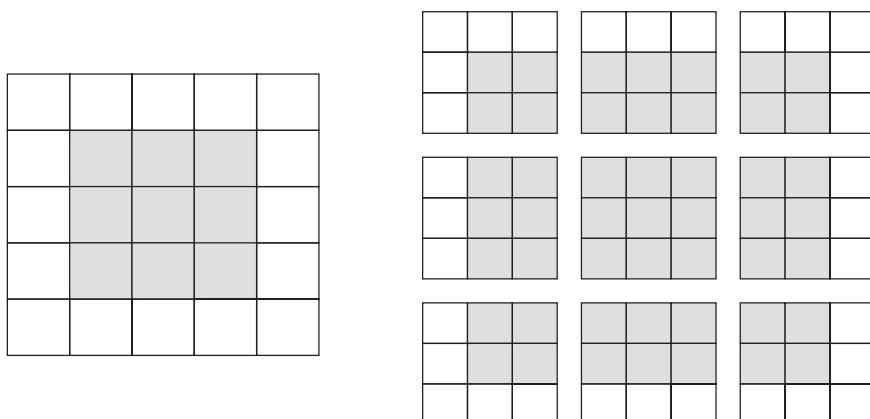
Обратите внимание на то, что выходные ширина и высота могут отличаться от входных. На то есть две причины:

- ❑ эффекты границ, которые могут устраняться дополнением входной карты признаков;
- ❑ использование *шага свертки*, определение которого приводится чуть ниже.

Рассмотрим подробнее эти понятия.

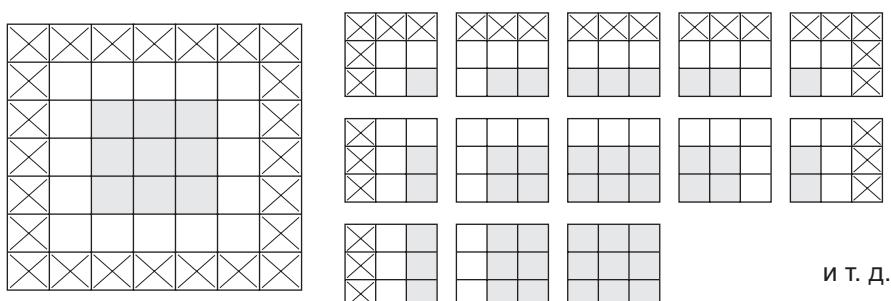
## Эффекты границ и дополнение

Рассмотрим карту признаков  $5 \times 5$  (всего 25 клеток). Существует всего 9 клеток, в которых может находиться центр окна  $3 \times 3$ , образующих сетку  $3 \times 3$  (рис. 5.5). Следовательно, карта выходных признаков будет иметь размер  $3 \times 3$ . Она получилась немного сжатой: ровно на две клетки вдоль каждого измерения. Вы можете увидеть, как проявляется эффект границ на более раннем примере: изначально у нас имелось  $28 \times 28$  входов, количество которых после первого сверточного слоя сократилось до  $26 \times 26$ .



**Рис. 5.5.** Допустимые местоположения шаблонов  $3 \times 3$  во входной карте признаков  $5 \times 5$

Чтобы получить выходную карту признаков с теми же пространственными размерами, что и входная карта, можно использовать *дополнение* (padding). Дополнение заключается в добавлении соответствующего количества строк и столбцов с каждой стороны входной карты признаков, чтобы можно было поместить центр окна свертки в каждую входную клетку. Для окна  $3 \times 3$  нужно добавить один столбец справа, один столбец слева, одну строку сверху и одну строку снизу. Для окна  $5 \times 5$  нужно добавить две строки (рис. 5.6).

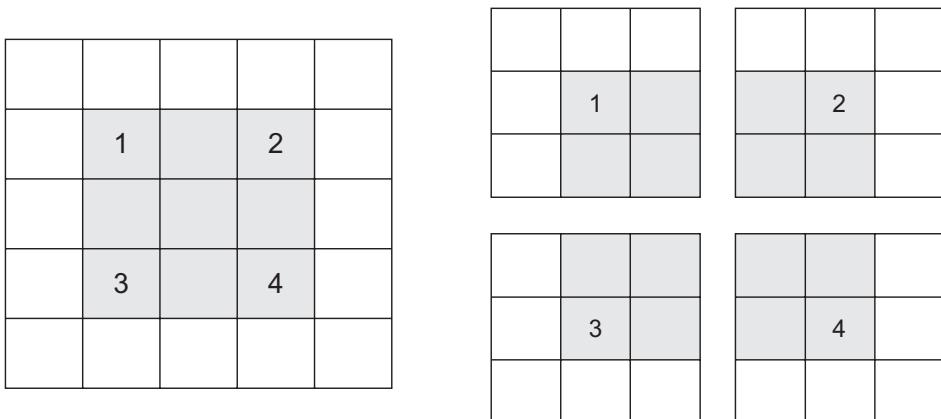


**Рис. 5.6.** Дополнение входной карты признаков  $5 \times 5$ , чтобы получить 25 шаблонов  $3 \times 3$

При использовании слоев Conv2D дополнение настраивается с помощью аргумента `padding`, который принимает два значения: "valid", означающее отсутствие дополнения (будут использоваться только допустимые местоположения окна), и "same", означающее «дополнить так, чтобы выходная карта признаков имела ту же ширину и высоту, что и входная». По умолчанию аргумент `padding` получает значение "valid".

## Шаг свертки

Другой фактор, который может влиять на размер выходной карты признаков, — *шаг свертки*. До сих пор в объяснениях выше предполагалось, что центральная клетка окна свертки последовательно перемещается в смежные клетки входной карты. Однако в общем случае расстояние между двумя соседними окнами является настраиваемым параметром, который называется *шагом свертки* и по умолчанию равен 1. Также имеется возможность определять *свертки с пробелами* (strided convolutions) — свертки с шагом больше 1. На рис. 5.7 можно видеть, как извлекаются шаблоны  $3 \times 3$  сверткой с шагом 2 из входной карты  $5 \times 5$  (без дополнения).



**Рис. 5.7.** Шаблоны  $3 \times 3$  свертки с шагом  $2 \times 2$

Использование шага 2 означает уменьшение ширины и высоты карты признаков за счет уменьшения разрешения в два раза (в дополнение к любым изменениям, вызванным эффектами границ). Свертки с пробелами редко используются на практике, хотя могут пригодиться в моделях некоторых типов, поэтому желательно знать и помнить об этой возможности.

Для уменьшения разрешения карты признаков вместо шага часто используется операция *выбора максимального значения из соседних* (max-pooling), которую вы видели в примере первой сверточной нейронной сети. Рассмотрим ее подробнее.

### 5.1.2. Выбор максимального значения из соседних (max-pooling)

В примере сверточной нейронной сети вы могли заметить, что размер карты признаков уменьшается вдвое после каждого слоя MaxPooling2D. Например, перед первым слоем MaxPooling2D карта признаков имела размер  $26 \times 26$ , но операция выбора максимального значения из соседних уменьшила ее до размера  $13 \times 13$ . В этом заключается предназначение данной операции: агрессивное уменьшение разрешения карты признаков, во многом подобное свертке с пробелами.

Операция выбора максимального значения из соседних заключается в следующем: из входной карты признаков извлекается окно, и из него выбирается максимальное значение для каждого канала. Концептуально это напоминает свертку, но вместо преобразования локальных шаблонов с обучением на линейных преобразованиях (ядро свертки) они преобразуются с использованием жестко заданной тензорной операции выбора максимального значения. Главное отличие от свертки состоит в том, что выбор максимального значения из соседних обычно производится с окном  $2 \times 2$  и шагом 2, чтобы уменьшить разрешение карты признаков в два раза. Собственно свертка, напротив, обычно выполняется с окном  $3 \times 3$  и без шага (шаг равен 1).

С какой целью вообще производится снижение разрешения карты признаков? Почему бы просто не убрать слой MaxPooling2D и не использовать карты признаков большего размера? Рассмотрим этот вариант. Сверточная основа модели в этом случае будет выглядеть так:

```
model_no_max_pool = models.Sequential()
model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu',
                                    input_shape=(28, 28, 1)))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Сводная информация о модели:

```
>>> model_no_max_pool.summary()
Layer (type)                  Output Shape           Param #
=====
conv2d_4 (Conv2D)             (None, 26, 26, 32)    320
conv2d_5 (Conv2D)             (None, 24, 24, 64)   18496
conv2d_6 (Conv2D)             (None, 22, 22, 64)   36928
=====
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

Что не так в этой конфигурации? Две вещи:

- ❑ Она не способствует изучению пространственной иерархии признаков. Окна  $3 \times 3$  в третьем слое содержат только информацию, поступающую из окон  $7 \times 7$  в исходных данных. Высокоуровневые шаблоны, изученные с помощью сверточной нейронной сети, будут слишком малы в сравнении с начальными данными, чего может оказаться недостаточно для обучения классификации цифр (попробуйте распознать цифру, посмотрев на нее через окна  $7 \times 7$  пикселов!). Нам нужно, чтобы признаки, полученные от последнего сверточного слоя, содержали информацию о совокупности исходных данных.
- ❑ Заключительная карта признаков имеет  $22 \times 22 \times 64 = 30\,976$  коэффициентов на образец. Это очень много. Если бы вы решили сделать ее плоской, чтобы наложить сверху слой Dense размером 512, этот слой имел бы 15,8 миллиона параметров. Это слишком много для такой маленькой модели и в результате приведет к интенсивному переобучению.

Проще говоря, уменьшение разрешения используется для уменьшения количества коэффициентов в карте признаков для обработки, а также внедрения иерархий пространственных фильтров путем создания последовательных слоев свертки для просмотра все более крупных окон (с точки зрения долей исходных данных, которые они охватывают).

Обратите внимание на то, что операция выбора максимального значения не единственный способ уменьшения разрешения. Как вы уже знаете, на предыдущих сверточных слоях можно также использовать шаг свертки. Кроме того, вместо выбора максимального значения вы можете использовать операцию выбора среднего значения по соседним элементам (average pooling), когда каждый локальный шаблон преобразуется путем взятия среднего значения для каждого канала в шаблоне вместо максимального. Однако операция выбора максимального значения обычно дает лучшие результаты, чем эти альтернативные решения. Причина в том, что признаки, как правило, кодируют пространственное присутствие некоторого шаблона или понятия в разных клетках карты признаков, поэтому *максимальное присутствие* признаков намного информативнее, чем *среднее присутствие*. Поэтому более разумная стратегия снижения разрешения состоит в том, чтобы сначала получить плотные карты признаков (путем обычной свертки без пробелов), а затем рассмотреть максимальные значения признаков в небольших шаблонах, а не разреженные окна из входных данных (путем свертки с пробелами) или усредненные шаблоны, которые могут привести к пропуску информации о присутствии.

На данном этапе у вас должно сложиться достаточно полное представление об основах сверточных нейронных сетей — картах признаков, операциях свертки и выбора максимального значения по соседним элементам, а также о том, как сконструировать небольшую сверточную нейронную сеть для решения такой простой задачи, как классификация цифр из набора MNIST. Теперь перейдем к более полезным и практическим применениям.

## 5.2. Обучение сверточной нейронной сети с нуля на небольшом наборе данных

Необходимость обучения модели классификации изображений на очень небольшом объеме данных — обычная ситуация, с которой вы наверняка столкнетесь в своей практике, если будете заниматься распознаванием образов с помощью технологий компьютерного зрения на профессиональном уровне. Под «небольшим» объемом понимается от нескольких сотен до нескольких десятков тысяч изображений. В качестве практического примера рассмотрим классификацию изображений собак и кошек из набора данных, содержащего 4000 изображений (2000 кошек, 2000 собак). Мы будем использовать 2000 изображений для обучения, 1000 для проверки и 1000 для контроля.

В этом разделе рассматривается одна простая стратегия решения данной задачи: обучение новой модели с нуля при наличии небольшого объема исходных данных. Сначала мы обучим маленькую сверточную нейронную сеть на 2000 обучающих образцах без применения регуляризации, чтобы задать базовый уровень достижимого. Она даст нам точность классификации 71 %. С этого момента начнет проявляться эффект переобучения. Затем вашему вниманию будет представлен эффективный способ уменьшения степени переобучения в распознавании образов — *расширение данных* (data augmentation). С его помощью мы повысим точность классификации до 82 %.

В следующем разделе мы рассмотрим еще два основных приема глубокого обучения на небольших наборах данных: *выделение признаков с использованием предварительно обученной сети* (поможет поднять точность с 90 до 96 %) и *дообучение предварительно обученной сети* (поможет достичь окончательной точности в 97 %). Вместе эти три стратегии — обучение малой модели с нуля, выделение признаков с использованием предварительно обученной модели и дообучение этой модели — станут вашим основным набором инструментов для решения задач классификации изображений с обучением на небольших наборах данных.

### 5.2.1. Целесообразность глубокого обучения для решения задач с небольшими наборами данных

Иногда можно услышать, что глубокое обучение можно применять только при наличии большого объема данных. Это утверждение верно лишь отчасти: одна из основных характеристик глубокого обучения — возможность самостоятельно находить информативные признаки в обучающих данных, без конструирования признаков вручную, а это достижимо только при наличии большого объема обучающих примеров. Это особенно верно для задач, когда входные образцы имеют много измерений, как, например, изображения.

Однако понятие «большой объем данных» весьма относительно, в первую очередь относительно размера и глубины обучаемой сети. Нельзя обучить сверточную

нейронную сеть решению сложной задачи на нескольких десятках образцов, а вот нескольких сотен вполне может хватить, если модель невелика и регуляризована, а решаемая задача проста. Так как сверточные нейронные сети изучают локальные признаки, инвариантные в отношении переноса, они обладают высокой эффективностью в решении задач распознавания. Обучение сверточной нейронной сети с нуля на очень небольшом наборе изображений дает вполне неплохие результаты, несмотря на относительную нехватку данных, без необходимости конструировать признаки вручную. В данном разделе мы убедимся в этом на практике.

Более того, модели глубокого обучения по своей природе очень гибкие: можно, к примеру, обучить модель для классификации изображений или распознавания речи на очень большом наборе данных и затем использовать ее для решения самых разных задач с небольшими модификациями. В частности, в распознавании образов многие предварительно обученные модели (обычно на наборе данных ImageNet) теперь доступны всем желающим для загрузки и могут использоваться как основа для создания очень мощных моделей распознавания образов на небольших объемах данных. Именно так мы и поступим в следующем разделе. Начнем с получения данных.

## 5.2.2. Загрузка данных

Набор данных «Dogs vs. Cats», который мы будем использовать, не поставляется в составе Keras. Он был создан в ходе состязаний по распознаванию образов в конце 2013-го, когда сверточные нейронные сети еще не заняли лидирующего положения, и доступен на сайте Kaggle. Этот набор можно получить по адресу: [www.kaggle.com/c/dogs-vs-cats/data](http://www.kaggle.com/c/dogs-vs-cats/data) (для этого вам потребуется создать учетную запись на сайте Kaggle, если у вас ее еще нет, но не волнуйтесь, процесс регистрации очень прост).

Этот набор содержит изображения в формате JPEG с низким разрешением. На рис. 5.8 показано несколько примеров.

Неудивительно, что состязание по классификации изображений кошек и собак на сайте Kaggle в 2013 году выиграли участники, использовавшие сверточные нейронные сети. Лучшие результаты достигали точности в 95 %. В этом примере мы приблизимся к этой точности (в следующем разделе), даже при том, что для обучения моделей будем использовать менее 10% данных, которые были доступны участникам состязаний.

Этот набор содержит 25 000 изображений кошек и собак (по 12 500 для каждого класса) общим объемом 543 Мбайт (в сжатом виде). После загрузки и распаковки архива мы создадим новый набор, разделенный на три поднабора: обучающий набор с 1000 образцами каждого класса, проверочный набор с 500 образцами каждого класса и контрольный набор с 500 образцами каждого класса.



**Рис. 5.8.** Примеры изображений из набора «Dogs vs. Cats». Размеры не были изменены: изображения имеют разные размеры, ракурсы съемки и т. д.

Все необходимое выполняет код в листинге 5.4.

**Листинг 5.4.** Копирование изображений в обучающий, проверочный и контрольный каталоги

```

Путь к каталогу с распакованным
исходным набором данных           Каталог для сохранения
                                         выделенного небольшого набора

import os, shutil

original_dataset_dir = '/Users/fchollet/Downloads/kaggle_original_data'

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'               ←

os.mkdir(base_dir)

train_dir = os.path.join(base_dir, 'train')                                ← Каталоги для
os.mkdir(train_dir)                                                       ← обучающего,
validation_dir = os.path.join(base_dir, 'validation')                      ← проверочного
os.mkdir(validation_dir)                                                 ← и контрольного
test_dir = os.path.join(base_dir, 'test')                                     ← поднаборов
os.mkdir(test_dir)

train_cats_dir = os.path.join(train_dir, 'cats') | Каталог для обучающих
os.mkdir(train_cats_dir) | изображений с кошками

train_dogs_dir = os.path.join(train_dir, 'dogs') | Каталог для обучающих
os.mkdir(train_dogs_dir) | изображений с собаками

```

Продолжение ↗

## Листинг 5.4 (продолжение)

```

validation_cats_dir = os.path.join(validation_dir, 'cats') | Каталог для про-
os.mkdir(validation_cats_dir) | верочных изобра-
| жений с кошками

validation_dogs_dir = os.path.join(validation_dir, 'dogs') | Каталог для про-
os.mkdir(validation_dogs_dir) | верочных изобра-
| жений с собаками

test_cats_dir = os.path.join(test_dir, 'cats') | Каталог для контрольных
os.mkdir(test_cats_dir) | изображений с кошками

test_dogs_dir = os.path.join(test_dir, 'dogs') | Каталог для контрольных
os.mkdir(test_dogs_dir) | изображений с собаками

fnames = ['cat.{}.jpg'.format(i) for i in range(1000)] | Копирование первых
for fname in fnames: | 1000 изображений
    src = os.path.join(original_dataset_dir, fname) | с кошками в каталог
    dst = os.path.join(train_cats_dir, fname) | train_cats_dir
    shutil.copyfile(src, dst)

fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)] | Копирование сле-
for fname in fnames: | дующих 500 изо-
    src = os.path.join(original_dataset_dir, fname) | бражений с кош-
    dst = os.path.join(validation_cats_dir, fname) | ками в каталог
    shutil.copyfile(src, dst) | validation_cats_dir

fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)] | Копирование сле-
for fname in fnames: | дующих 500 изо-
    src = os.path.join(original_dataset_dir, fname) | бражений с кош-
    dst = os.path.join(test_cats_dir, fname) | ками в каталог
    shutil.copyfile(src, dst) | test_cats_dir

fnames = ['dog.{}.jpg'.format(i) for i in range(1000)] | Копирование первых
for fname in fnames: | 1000 изображений
    src = os.path.join(original_dataset_dir, fname) | с собаками в каталог
    dst = os.path.join(train_dogs_dir, fname) | train_dogs_dir
    shutil.copyfile(src, dst)

fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)] | Копирование сле-
for fname in fnames: | дующих 500 изо-
    src = os.path.join(original_dataset_dir, fname) | бражений с соба-
    dst = os.path.join(validation_dogs_dir, fname) | ками в каталог
    shutil.copyfile(src, dst) | validation_dogs_dir

fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)] | Копирование сле-
for fname in fnames: | дующих 500 изо-
    src = os.path.join(original_dataset_dir, fname) | бражений с соба-
    dst = os.path.join(test_dogs_dir, fname) | ками в каталог
    shutil.copyfile(src, dst) | test_dogs_dir

```

Для проверки подсчитаем, сколько изображений оказалось в каждом поднаборе (обучающем/проверочном/контрольном):

```
>>> print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000
>>> print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000
>>> print('total validation cat images:', len(os.listdir(validation_cats_
dir)))
total validation cat images: 500
>>> print('total validation dog images:', len(os.listdir(validation_dogs_
dir)))
total validation dog images: 500
>>> print('total test cat images:', len(os.listdir(test_cats_dir)))
total test cat images: 500
>>> print('total test dog images:', len(os.listdir(test_dogs_dir)))
total test dog images: 500
```

Итак, у нас действительно имеется 2000 обучающих, 1000 проверочных и 1000 контрольных изображений. Каждый поднабор содержит одинаковое количество образцов каждого класса: это сбалансированная задача бинарной классификации, соответственно, мерой успеха может служить точность классификации.

### 5.2.3. Конструирование сети

В предыдущем примере мы сконструировали небольшую сверточную нейронную сеть для данных MNIST, и теперь вы знаете, что это такое. В этом примере мы реализуем ту же самую общую структуру: сверточная нейронная сеть будет организована как стек чередующихся слоев Conv2D (с функцией активации `relu`) и `MaxPooling2D`.

Однако, так как мы имеем дело с большими изображениями и решаем более сложную задачу, мы сделаем сеть больше: она будет иметь на одну пару слоев `Conv2D + MaxPooling2D` больше. Это увеличит ее емкость и обеспечит дополнительное снижение размеров карт признаков, чтобы они не оказались слишком большими, когда достигнут слоя `Flatten`. С учетом того, что мы начнем с входов, имеющих размер  $150 \times 150$  (выбор был сделан совершенно произвольно), в конце, точно перед слоем `Flatten`, получится карта признаков размером  $7 \times 7$ .

#### ПРИМЕЧАНИЕ

Глубина карт признаков в сети будет постепенно увеличиваться (с 32 до 128), а их размеры — уменьшаться (со  $148 \times 148$  до  $7 \times 7$ ). Этот шаблон вы будете видеть почти во всех сверточных нейронных сетях.

Так как перед нами стоит задача бинарной классификации, сеть должна заканчиваться единственным признаком (слой `Dense` с размером 1 и функцией активации `sigmoid`). Этот признак будет представлять собой вероятность принадлежности рассматриваемого изображения одному из двух классов.

**Листинг 5.5.** Создание небольшой сверточной нейронной сети для классификации изображений кошек и собак

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))

model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Посмотрим, как изменяются размеры карт признаков с каждым последующим слоем:

>>> model.summary()		
Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 148, 148, 32)	896
maxpooling2d_1 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_2 (Conv2D)	(None, 72, 72, 64)	18496
maxpooling2d_2 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_3 (Conv2D)	(None, 34, 34, 128)	73856
maxpooling2d_3 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_4 (Conv2D)	(None, 15, 15, 128)	147584
maxpooling2d_4 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 512)	3211776
dense_2 (Dense)	(None, 1)	513

Total params: 3,453,121  
Trainable params: 3,453,121  
Non-trainable params: 0

На этапе компиляции, как обычно, используем оптимизатор `RMSprop`. Так как сеть заканчивается единственным признаком, используем функцию потерь `binary_crossentropy` (для напоминания: в табл. 4.1 приводится шпаргалка по использованию разных функций потерь в разных ситуациях).

#### **Листинг 5.6.** Настройка модели для обучения

```
from keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

### 5.2.4. Предварительная обработка данных

Как вы уже знаете, перед передачей в сеть данные должны быть преобразованы в тензоры с вещественными числами. В настоящее время данные хранятся в виде файлов JPEG, поэтому их нужно подготовить для передачи в сеть, выполнив следующие шаги:

1. Прочитать файлы с изображениями.
2. Декодировать содержимое из формата JPEG в таблицы пикселов RGB.
3. Преобразовать их в тензоры с вещественными числами.
4. Масштабировать значения пикселов из диапазона [0, 255] в диапазон [0, 1] (как вы уже знаете, нейронным сетям предпочтительнее передавать небольшие значения).

Этот порядок действий может показаться немного сложным, но, к счастью, в Keras имеются утилиты, способные выполнить его автоматически. Во фреймворке Keras имеется модуль `keras.preprocessing.image` с инструментами для обработки изображений. В частности, в нем вы найдете класс `ImageDataGenerator`, который позволит быстро настроить генераторы Python для автоматического преобразования файлов с изображениями в пакеты готовых тензоров. В листинге 5.7 показано, как им воспользоваться.

#### **Листинг 5.7.** Использование `ImageDataGenerator` для чтения изображений из каталогов

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255) | Масштабировать значения
test_datagen = ImageDataGenerator(rescale=1./255) | с коэффициентом 1/255

train_generator = train_datagen.flow_from_directory(
    train_dir, ← Целевой каталог
    target_size=(150, 150), ← Привести все изображения к размеру 150 × 150
    batch_size=20,
```

Продолжение ↗

**Листинг 5.7** (продолжение)

```
class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')
```

Так как используется функция потерь binary\_crossentropy, метки должны быть бинарными

## Генераторы Python

*Генератором* в языке Python называется объект, действующий как итератор: его можно использовать с инструкцией цикла `for ... in`. Работа генераторов основана на использовании инструкции `yield`.

Вот пример генератора, возвращающего целые числа:

```
def generator():
    i = 0
    while True:
        i += 1
        yield i

for item in generator():
    print(item)
    if item > 4:
        break
```

На экран будет выведено следующее:

```
1
2
3
4
5
```

Рассмотрим вывод одного из таких генераторов: он возвращает пакеты изображений  $150 \times 150$  в формате RGB (с формой  $(20, 150, 150, 3)$ ) и бинарные метки (с формой  $(20,)$ ). В каждом пакете имеется 20 образцов (размер пакета). Обратите внимание на то, что этот генератор возвращает пакеты до бесконечности: он выполняет бесконечный цикл перебора изображений в целевом каталоге. По этой причине мы должны прервать цикл в некоторый момент:

```
>>> for data_batch, labels_batch in train_generator:
>>>     print('data batch shape:', data_batch.shape)
>>>     print('labels batch shape:', labels_batch.shape)
>>>     break
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Передадим исходные данные в модель с помощью генератора. Используем для этого метод `fit_generator`, эквивалент метода `fit` для генераторов данных, подобных этому. В первом аргументе он принимает генератор Python, бесконечно возвращающий

пакеты входных данных и целей, как это делает данный генератор. Так как данные генерируются до бесконечности, модель Keras должна знать, сколько образцов извлечь из генератора, прежде чем объявить эпоху завершенной. Этую функцию выполняет аргумент `steps_per_epoch`: после извлечения `steps_per_epoch` пакетов из генератора, то есть после выполнения `steps_per_epoch` шагов градиентного спуска, процесс обучения переходит к следующей эпохе. В данном случае пакеты содержат по 20 образцов, поэтому для получения 2000 образцов модель извлекает 100 пакетов.

При использовании метода `fit_generator` вы можете передать аргумент `validation_data`, так же как методу `fit`. Важно отметить, что этот аргумент может быть не только генератором данных, но также кортежем массивов NumPy. Если в `validation_data` передать генератор, предполагается, что он будет возвращать пакеты проверочных данных до бесконечности; поэтому вы должны также передать аргумент `validation_steps`, определяющий количество пакетов, извлекаемых из генератора проверочных данных для оценки.

#### **Листинг 5.8.** Обучение модели с использованием генератора пакетов

```
history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50)
```

Хорошей практикой считается всегда сохранять модели после обучения.

#### **Листинг 5.9.** Сохранение модели

```
model.save('cats_and_dogs_small_1.h5')
```

Создадим графики изменения точности и потерь модели по обучающим и проверочным данным в процессе обучения (рис. 5.9 и 5.10).

#### **Листинг 5.10.** Формирование графиков изменения потерь и точности в процессе обучения

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

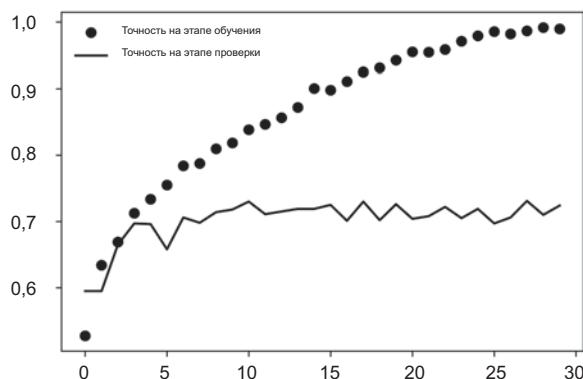
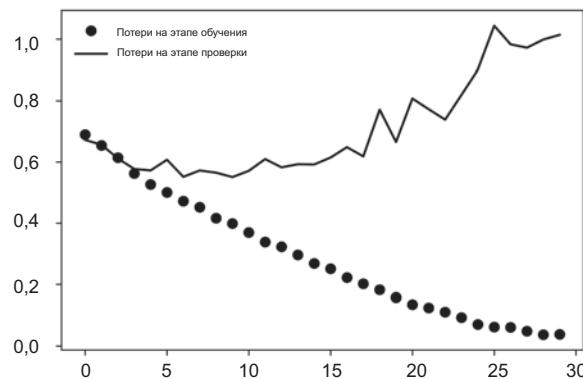
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()
```

**Листинг 5.10** (продолжение)

```
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

**Рис. 5.9.** Точность на этапах обучения и проверки**Рис. 5.10.** Потери на этапах обучения и проверки

На графиках четко наблюдается эффект переобучения. Точность на обучающих данных линейно растет и приближается к 100 %, тогда как точность на проверочных данных останавливается на отметке 70–72 %. Потери на этапе проверки достигают минимума всего после пяти эпох и затем замирают, а потери на этапе обучения продолжают линейно уменьшаться, почти достигая 0.

Поскольку у нас относительно немного обучающих образцов (2000), переобучение становится проблемой номер один. Вы уже знаете несколько методов, помогающих смягчить эту проблему, таких как прореживание и сокращение весов

(L2-регуляризация). Теперь мы познакомимся с еще одним способом, характерным для распознавания образов и используемым почти повсеместно при обработке изображений с применением моделей глубокого обучения: способом *расширения данных* (data augmentation).

### 5.2.5. Расширение данных

Причиной переобучения является недостаточное количество образцов для обучения модели, способной обобщать новые данные. Имея бесконечный объем данных, можно было бы получить модель, учитывающую все аспекты распределения данных: эффект переобучения никогда не наступил бы. Прием расширения данных реализует подход создания дополнительных обучающих данных из имеющихся путем трансформации образцов множеством случайных преобразований, дающих правдоподобные изображения. Цель состоит в том, чтобы на этапе обучения модель никогда не увидела одно и то же изображение дважды. Это поможет модели выявить больше особенностей данных и достичь лучшей степени обобщения.

Сделать это в Keras можно путем настройки ряда случайных преобразований для изображений, читаемых экземпляром `ImageDataGenerator`. Начнем с простого примера.

#### Листинг 5.11. Настройка расширения данных в `ImageDataGenerator`

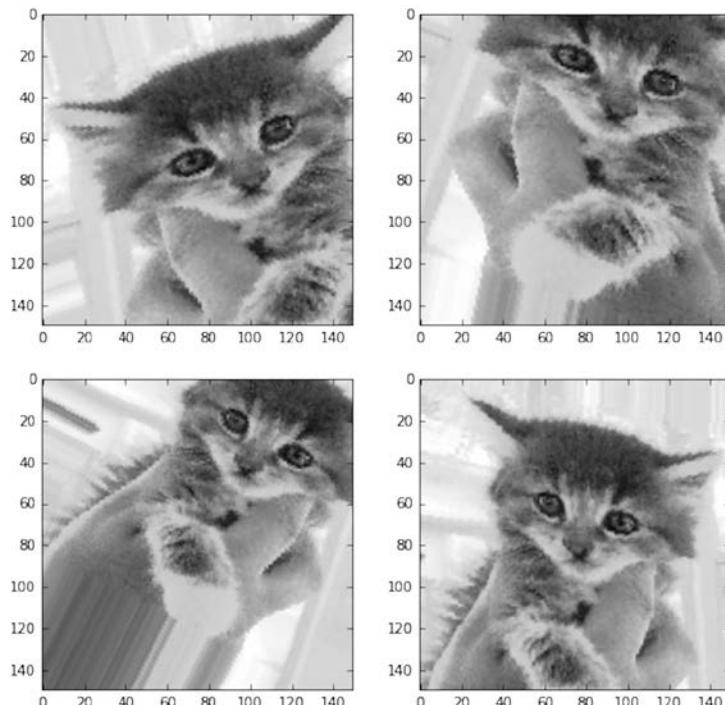
```
datagen = ImageDataGenerator(  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

Здесь представлена лишь часть возможных вариантов (полный список вы найдете в документации к фреймворку Keras). Давайте быстро пробежимся по этому коду:

- ❑ `rotation_range` — величина в градусах (0–180), диапазон, в котором будет осуществляться случайный поворот изображения;
- ❑ `width_shift` и `height_shift` — диапазоны (в долях ширины и высоты), в пределах которых изображения смещаются по горизонтали и вертикали соответственно;
- ❑ `shear_range` — для случайного применения сдвигового (shearing) преобразования;
- ❑ `zoom_range` — для случайного изменения масштаба внутри изображений;

- `horizontal_flip` — для случайного переворачивания половины изображения по горизонтали — подходит в случае отсутствия предположений о горизонтальной асимметрии (например, в изображениях реального мира);
- `fill_mode` — стратегия заполнения вновь созданных пикселов, появляющихся после поворота или смещения по горизонтали/вертикали.

Взгляните на дополнительные изображения (рис. 5.11).



**Рис. 5.11.** Изображения с кошкой, полученные применением случайных расширений

**Листинг 5.12.** Отображение некоторых обучающих изображений, подвергшихся случайным преобразованиям

```
from keras.preprocessing import image
fnames = [os.path.join(train_cats_dir, fname) for
          fname in os.listdir(train_cats_dir)]
img_path = fnames[3]
img = image.load_img(img_path, target_size=(150, 150))
x = image.img_to_array(img)
x = x.reshape((1,) + x.shape)
```

Модуль с утилитами для обработки изображений

Выбор одного изображения для расширения

Чтение изображения и изменение его размеров

Преобразование в массив Numpy с формой (150, 150, 3)

Изменение формы на (1, 150, 150, 3)

```
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()
```

Генерация пакетов случайно преобразованных изображений. Цикл выполняется бесконечно, поэтому его нужно принудительно прервать в какой-то момент!

Если обучить новую сеть с использованием этих настроек расширения данных, она никогда не увидит одно и то же изображение дважды. Однако входные данные по-прежнему будут тесно связаны между собой, потому что получены из небольшого количества оригинальных изображений, — у вас не получится сгенерировать новую информацию, вы можете только повторить существующую. Поэтому данного решения недостаточно, чтобы избавиться от эффекта переобучения. Продолжая борьбу с ним, добавим в модель слой `Dropout` непосредственно перед полносвязанным классификатором.

**Листинг 5.13.** Определение новой сверточной нейронной сети, включающей в себя слой прореживания

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Обучим сеть, задействовав расширение данных и прореживание.

**Листинг 5.14.** Обучение сверточной нейронной сети с использованием генераторов расширения данных

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
```

Продолжение ↗

**Листинг 5.14** (продолжение)

```

shear_range=0.2,
zoom_range=0.2,
horizontal_flip=True,)

test_datagen = ImageDataGenerator(rescale=1./255) ←

train_generator = train_datagen.flow_from_directory(
    train_dir, ← Целевой каталог
    target_size=(150, 150), ← Приведение всех изображений к размеру 150 × 150
    batch_size=32,
    class_mode='binary') ←

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary') ←

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50)

```

Обратите внимание,  
что проверочные  
данные не требуется  
расширять!

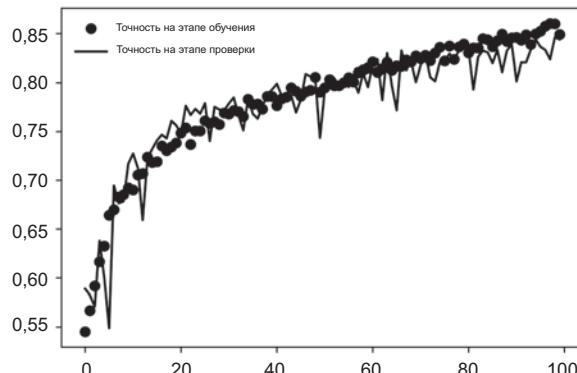
Так как используется  
функция потерь  
`binary_crossentropy`,  
метки должны быть  
бинарными

А теперь сохраним модель — мы будем использовать ее в разделе 5.4.

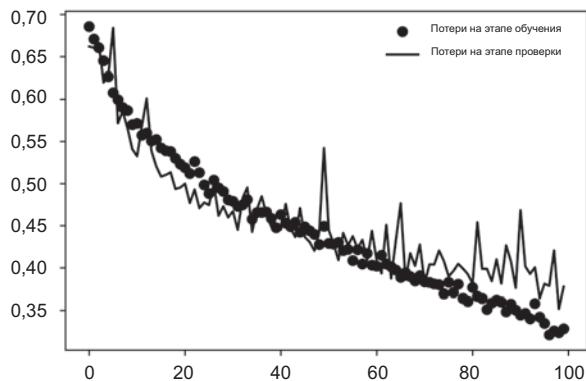
**Листинг 5.15.** Сохранение модели

```
model.save('cats_and_dogs_small_2.h5')
```

И снова выведем графики с результатами (рис. 5.12 и 5.13). Применив расширение данных и прореживание, мы практически избавились от переобучения: кривые точности и потерь на этапе обучения близки к аналогичным кривым на этапе проверки. Теперь мы достигли точности 82 %, улучшив ее на 15 %, в сравнении с нерегуляризованной моделью.



**Рис. 5.12.** Точность на этапах проверки и обучения с расширением данных



**Рис. 5.13.** Потери на этапах проверки и обучения с расширением данных

Используя дополнительные методы регуляризации и настроив параметры сети (например, число фильтров на сверточный слой или число слоев в сети), можно добиться еще более высокой точности, примерно 86 % или 87 %. Однако будет очень трудно подняться выше этой отметки, обучая сверточную нейронную сеть с нуля, потому что у нас слишком мало данных. Следующий шаг к увеличению точности решения этой задачи заключается в использовании предварительно обученной модели, но об этом мы поговорим в следующих двух разделах.

## 5.3. Использование предварительно обученной сверточной нейронной сети

Типичным и эффективным подходом к глубокому обучению на небольших наборах изображений является использование предварительно обученной сети. *Предварительно обученная сеть* — это сохраненная сеть, прежде обученная на большом наборе данных, обычно в рамках масштабной задачи классификации изображений. Если этот исходный набор данных достаточно велик и достаточно обобщен, тогда пространственная иерархия признаков, изученных сетью, может эффективно выступать в роли обобщенной модели видимого мира и быть полезной во многих разных задачах распознавания образов, даже если эти новые задачи будут связаны с совершенно иными классами, отличными от классов в оригинальной задаче. Другими словами, можно обучить сеть на изображениях из ImageNet (где подавляющее большинство классов — животные и бытовые предметы) и затем использовать эту обученную сеть для идентификации чего-то иного, например предметов мебели на изображениях. Такая переносимость изученных признаков между разными задачами — главное преимущество глубокого обучения перед многими более старыми приемами поверхностного обучения, которое делает глубокое обучение очень эффективным инструментом для решения задач с малым объемом данных.

В нашем случае мы возьмем за основу сверточную нейронную сеть, обученную на наборе ImageNet (1,4 миллиона изображений, классифицированных на 1000 разных классов). Коллекция ImageNet содержит множество изображений разных животных, включая разновидности кошек и собак, а значит, можно рассчитывать, что модель, обученная на этой коллекции, прекрасно справится с нашей задачей классификации изображений кошек и собак.

Мы воспользуемся архитектурой VGG16, разработанной Кареном Симоняном (Karen Simonyan) и Эндрю Циссерманом (Andrew Zisserman) в 2014-м; это простая и широко используемая архитектура сверточной нейронной сети для обучения на коллекции ImageNet<sup>1</sup>. Хотя это довольно старая модель, далеко отставшая от современного уровня, которая к тому же немного тяжелее многих более современных моделей, я выбрал ее, потому что ее архитектура похожа на примеры, представленные в этой книге выше, и вам будет проще понять ее без знакомства с какими-либо новыми понятиями. Возможно, это ваша первая встреча с одним из представителей всех этих моделей, названия которых вызывают дрожь, — VGG, ResNet, Inception, Inception-ResNet, Xception и т. д.; но со временем вы привыкнете к ним, потому что они часто будут встречаться на вашем пути, если вы продолжите заниматься применением глубокого обучения в распознавании образов.

Есть два приема использования предварительно обученных сетей: *выделение признаков* (feature extraction) и *дообучение* (fine-tuning). Мы рассмотрим оба и начнем с выделения признаков.

### 5.3.1. Выделение признаков

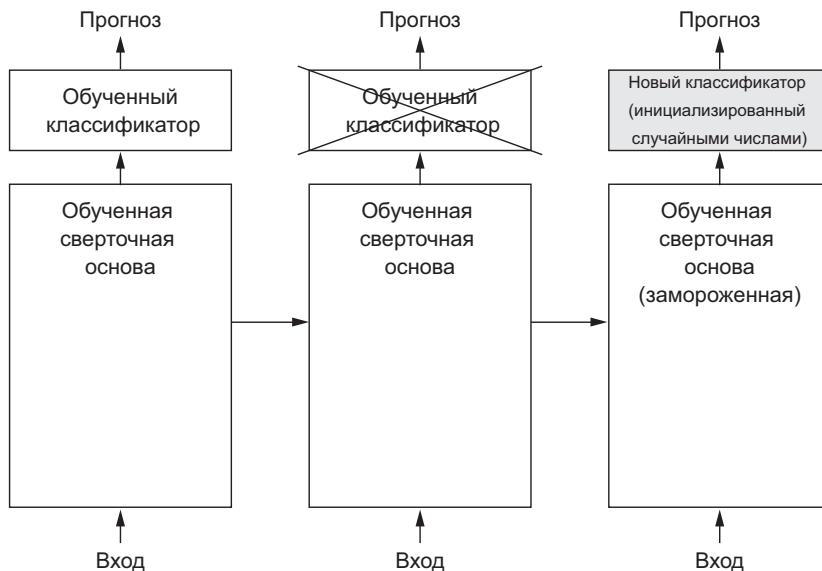
Выделение признаков заключается в использовании представлений, изученных предыдущей сетью, для выделения признаков из новых образцов, которые затем пропускаются через новый классификатор, обучаемый с нуля.

Как было показано выше, сверточные нейронные сети, используемые для классификации изображений, состоят из двух частей: они начинаются с последовательности слоев выбора значений и свертки и заканчиваются полносвязным классификатором. Первая часть называется *сверточной основой* (convolutional base) модели. В случае со сверточными нейронными сетями процесс выделения признаков заключается в том, чтобы взять сверточную основу предварительно обученной сети, пропустить через нее новые данные и на основе вывода обучить новый классификатор (рис. 5.14).

Почему повторно используется только сверточная основа? Нельзя ли повторно использовать полносвязный классификатор? В общем случае этого следует избегать. Причина в том, что представления, полученные сверточной основой, обычно более

---

<sup>1</sup> Karen Simonyan and Andrew Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition», arXiv (2014), <https://arxiv.org/abs/1409.1556>.



**Рис. 5.14.** Замена классификаторов при использовании одной и той же сверточной основы

универсальны, а значит, более пригодны для повторного использования: карты признаков сверточной нейронной сети — это карты присутствия на изображениях обобщенных понятий, которые могут пригодиться независимо от конкретной задачи распознавания образов. Но представления, изученные классификатором, обязательно будут характерны для набора классов, на котором обучалась модель, — они будут содержать только информацию о вероятности присутствия того или иного класса на изображении. Кроме того, представления, присутствующие в полно связных слоях, не содержат никакой информации *о местоположении* объекта на исходном изображении (эти слои лишены понятия пространства), тогда как сверточные карты признаков все еще хранят ее. Для задач, где местоположение объектов имеет значение, полно связные признаки почти бесполезны.

Отметим также, что уровень обобщенности (и, соответственно, пригодности к повторному использованию) представлений, выделенных конкретными сверточными слоями, зависит от глубины слоя в модели. Слои, следующие первыми, выделяют локальные, наиболее обобщенные карты признаков (таких, как визуальные границы, цвет и текстура), тогда как слои, располагающиеся дальше (или выше), выделяют более абстрактные понятия (такие, как «глаз кошки» или «глаз собаки»). Поэтому, если новый набор данных существенно отличается от набора, на котором обучалась оригинальная модель, возможно большего успеха можно добиться, если использовать только несколько первых слоев модели, а не всю сверточную основу.

В нашем случае, поскольку набор классов ImageNet содержит несколько классов кошек и собак, вероятно, было бы полезно повторно использовать информацию, содержащуюся в полносвязных слоях оригинальной модели. Но мы не будем этого делать, чтобы охватить более общий случай, когда набор классов из новой задачи не пересекается с набором классов оригинальной модели. А теперь перейдем к практике и используем сверточную основу сети VGG16, обученной на данных ImageNet, для выделения полезных признаков из изображений кошек и собак, а затем обучим классификатор кошек и собак, опираясь на эти признаки.

Модель VGG16 входит в состав фреймворка Keras. Ее можно импортировать из модуля `keras.applications`. Вот список моделей классификации изображений (все они предварительно обучены на наборе ImageNet), доступных в `keras.applications`:

- Xception
- Inception V3
- ResNet50
- VGG16
- VGG19
- MobileNet

Создадим экземпляр модели VGG16.

#### **Листинг 5.16.** Создание экземпляра сверточной основы VGG16

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

Здесь конструктору передаются три аргумента:

- Аргумент `weights` определяет источник весов для инициализации модели.
- Аргумент `include_top` определяет необходимость подключения к сети полносвязного классификатора. По умолчанию этот полносвязный классификатор соответствует 1000 классов из ImageNet. Так как мы намереваемся использовать свой полносвязный классификатор (только с двумя классами: `cat` и `dog`), мы не будем подключать его.
- Аргумент `input_shape` определяет форму тензоров с изображениями, которые будут подаваться на вход сети. Это необязательный аргумент: если опустить его, сеть сможет обрабатывать изображения любого размера.

Далее приводится информация о сверточной основе VGG16. Она напоминает простые сверточные нейронные сети, уже знакомые вам:

```
>>> conv_base.summary()
Layer (type)                  Output Shape                 Param #
=====
input_1 (InputLayer)          (None, 150, 150, 3)        0
block1_conv1 (Convolution2D)   (None, 150, 150, 64)      1792
block1_conv2 (Convolution2D)   (None, 150, 150, 64)      36928
block1_pool (MaxPooling2D)     (None, 75, 75, 64)        0
block2_conv1 (Convolution2D)   (None, 75, 75, 128)       73856
block2_conv2 (Convolution2D)   (None, 75, 75, 128)       147584
block2_pool (MaxPooling2D)     (None, 37, 37, 128)       0
block3_conv1 (Convolution2D)   (None, 37, 37, 256)       295168
block3_conv2 (Convolution2D)   (None, 37, 37, 256)       590080
block3_conv3 (Convolution2D)   (None, 37, 37, 256)       590080
block3_pool (MaxPooling2D)     (None, 18, 18, 256)       0
block4_conv1 (Convolution2D)   (None, 18, 18, 512)       1180160
block4_conv2 (Convolution2D)   (None, 18, 18, 512)       2359808
block4_conv3 (Convolution2D)   (None, 18, 18, 512)       2359808
block4_pool (MaxPooling2D)     (None, 9, 9, 512)        0
block5_conv1 (Convolution2D)   (None, 9, 9, 512)        2359808
block5_conv2 (Convolution2D)   (None, 9, 9, 512)        2359808
block5_conv3 (Convolution2D)   (None, 9, 9, 512)        2359808
block5_pool (MaxPooling2D)     (None, 4, 4, 512)        0
=====
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Заключительная карта признаков имеет форму  $(4, 4, 512)$ . Поверх нее мы положим полно связанный классификатор.

Далее можно пойти двумя путями:

- ❑ Пропустить наш набор данных через сверточную основу, записать получившийся массив Numpy на диск и затем использовать его как входные данные для

отдельного, полносвязного классификатора, похожего на тот, что мы видели в первой части книги. Это быстрое и незатратное решение, потому что требует запускать сверточную основу только один раз для каждого входного изображения, а сверточная основа — самая дорогостоящая часть конвейера. Однако по той же причине этот прием не позволит использовать прием расширения данных.

- Дополнить имеющуюся модель (`conv_base`) слоями `Dense` и пропустить все входные данные. Этот путь позволяет использовать расширение данных, потому что каждое изображение проходит через сверточную основу каждый раз, когда попадает в модель. Однако по той же причине этот путь намного более затратный, чем первый.

Мы охватим оба приема. Сначала рассмотрим код, реализующий первый прием: запись вывода `conv_base` в ответ на передачу наших данных и его использование в роли входных данных новой модели.

### Быстрое выделение признаков без расширения данных

Сначала запустим экземпляры представленного ранее класса `ImageDataGenerator`, чтобы извлечь изображения и их метки в массивы Numpy. Затем выделим признаки из этих изображений путем вызова метода `predict` модели `conv_base`.

**Листинг 5.17.** Выделение признаков с использованием предварительно обученной сверточной основы

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
    i += 1
    return features, labels
```

```

labels[i * batch_size : (i + 1) * batch_size] = labels_batch
i += 1
if i * batch_size >= sample_count:
    break
return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

```

Обратите внимание: поскольку генераторы возвращают данные в цикле до бесконечности, мы должны прервать цикл после передачи всех изображений

В настоящий момент выделенные признаки имеют форму (*образцы*, 4, 4, 512). Мы будем передавать их на вход полносвязного классификатора, поэтому мы должны привести этот тензор к форме (*образцы*, 8192):

```

train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))

```

Теперь можно определить свой полносвязный классификатор (обратите внимание на то, что для регуляризации здесь используется прием прореживания) и обучить его на только что записанных данных и метках.

#### **Листинг 5.18.** Определение и обучение полносвязного классификатора

```

from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

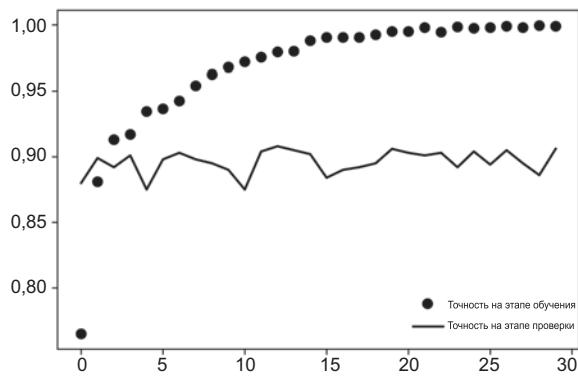
model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(train_features, train_labels,
                     epochs=30,
                     batch_size=20,
                     validation_data=(validation_features, validation_labels))

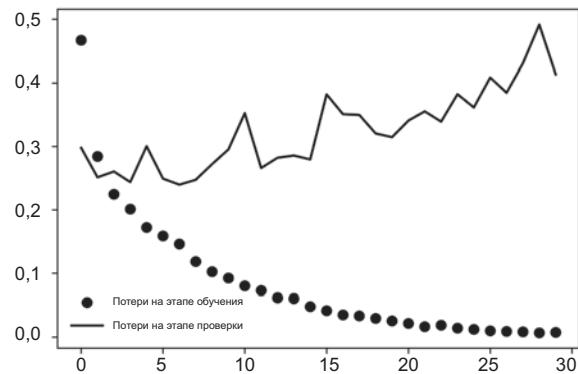
```

Обучение проходит очень быстро, потому что мы определили только два слоя `Dense`, — одна эпоха длится меньше одной секунды даже при выполнении на CPU.

Посмотрим теперь на графики изменения потерь и точности в процессе обучения (рис. 5.15 и 5.16).



**Рис. 5.15.** Точность на этапах проверки и обучения для простого извлечения признаков



**Рис. 5.16.** Потери на этапах проверки и обучения для простого извлечения признаков

### Листинг 5.19. Построение графиков

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
```

```
plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Мы достигли точности, близкой к 90 %, — более высокой, чем в предыдущем разделе, где обучали небольшую модель с нуля. Однако графики также показывают, что почти с самого начала стал проявляться эффект переобучения, несмотря на выбор довольно большого коэффициента прореживания. Это объясняется тем, что данный прием не использует расширение данных, которое необходимо для предотвращения переобучения на небольших наборах изображений.

## Выделение признаков с расширением данных

Теперь рассмотрим второй прием выделения признаков, более медленный и затратный, но позволяющий использовать расширение данных в процессе обучения, — прием расширения модели `conv_base` и ее использования для классификации.

### ПРИМЕЧАНИЕ

Этот прием настолько затратный, что его следует применять только при наличии GPU — он абсолютно не под силу CPU. Если у вас нет возможности запустить свой код на GPU, первый путь остается для вас единственным доступным решением.

Так как модели действуют подобно слоям, вы можете добавить модель (такую, как `conv_base`) в модель `Sequential` как самый обычный слой.

### Листинг 5.20. Добавление полносвязного классификатора поверх сверточной основы

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Вот как теперь выглядит модель:

```
>>> model.summary()
Layer (type)                  Output Shape                 Param #
=====

```

vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
dense_2 (Dense)	(None, 1)	257
<hr/> <hr/> <hr/>		
Total params: 16,812,353		
Trainable params: 16,812,353		
Non-trainable params: 0		

Как видите, сверточная основа VGG16 имеет 14 714 688 параметров, что представляет собой достаточно большое число. Классификатор, добавленный сверху, имеет 2 миллиона параметров.

Перед компиляцией и обучением модели очень важно заморозить сверточную основу. *Замораживание* одного или нескольких слоев предотвращает изменение весовых коэффициентов в них в процессе обучения. Если этого не сделать, тогда представления, прежде изученные сверточной основой, изменятся в процессе обучения на новых данных. Так как слои `Dense` сверху инициализируются случайными значениями, в сети могут произойти существенные изменения весов, фактически разрушив представления, полученные ранее.

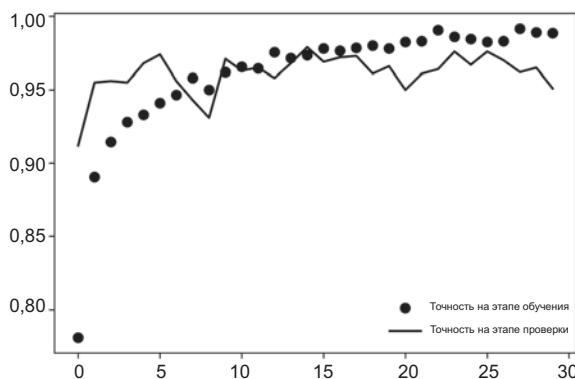
В Keras, чтобы заморозить сеть, нужно передать атрибут `trainable` со значением `False`:

```
>>> print('This is the number of trainable weights '
       'before freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights before freezing the conv base: 30
>>> conv_base.trainable = False
>>> print('This is the number of trainable weights '
       'after freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights after freezing the conv base: 4
```

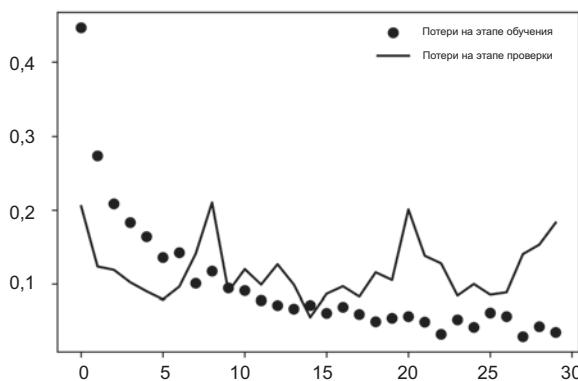
В этом случае обучению будут подвергаться только веса из двух вновь добавленных слоев `Dense`, то есть всего четыре весовых тензора: по два на слой (главная весовая матрица и вектор смещений). Обратите внимание: чтобы эти изменения вступили в силу, необходимо скомпилировать модель. Если признак обучения весов изменяется после компиляции модели, необходимо снова перекомпилировать модель, иначе это изменение будет игнорироваться.

Теперь можно начинать обучение модели. Используем те же настройки расширения данных, как в предыдущем примере.

Снова построим графики изменения потерь и точности в процессе обучения (рис. 5.17 и 5.18). Как видите, мы почти достигли точности в 96% на этапе проверки. Этот результат намного лучше, чем в случае обучения небольшой сверточной нейронной сети с нуля.



**Рис. 5.17.** Точность на этапах проверки и обучения для извлечения признаков с расширением данных



**Рис. 5.18.** Потери на этапах проверки и обучения для извлечения признаков с расширением данных

**Листинг 5.21.** Полное обучение модели с замороженной сверточной основой

```
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers
```

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

```
test_datagen = ImageDataGenerator(rescale=1./255)
```

Обратите внимание:  
проверочные  
данные не требуется  
расширять!

Продолжение ↗

**Листинг 5.21** (продолжение)

```

train_generator = train_datagen.flow_from_directory(
    train_dir, ← Целевой каталог
    target_size=(150, 150), ← Приведение всех изображений к размеру 150 × 150
    batch_size=20,
    class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary') ← Так как используется
                           функция потерь
                           binary_crossentropy,
                           метки должны быть
                           бинарными

model.compile(loss='binary_crossentropy',
               optimizer=optimizers.RMSprop(lr=2e-5),
               metrics=['acc'])

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50)

```

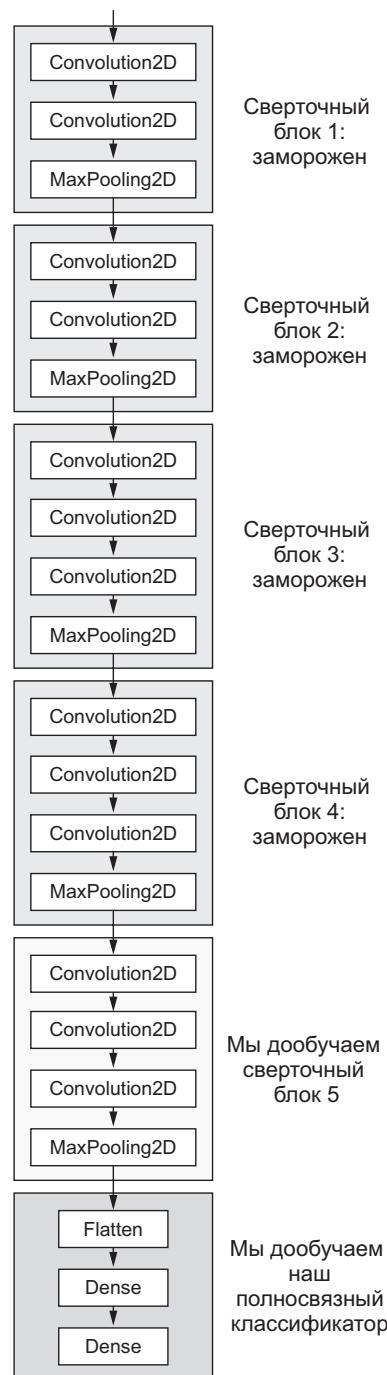
### 5.3.2. Дообучение

Другой широко используемый прием повторного использования модели, дополняющий выделение признаков, — *дообучение* (fine-tuning) (рис. 5.19). Дообучение заключается в размораживании нескольких верхних слоев замороженной модели, которая использовалась для выделения признаков, и совместном обучении вновь добавленной части модели (в данном случае полносвязного классификатора) и этих верхних слоев. Этот прием называется *дообучением*, поскольку немного корректирует наиболее абстрактные представления в повторно используемой модели, чтобы сделать их более актуальными для данной задачи.

Выше я отмечал, что необходимо заморозить сверточную основу сети VGG16, чтобы получить возможность обучить классификатор, инициализированный случайными значениями. По той же причине после обучения классификатора можно дообучить несколько верхних слоев сверточной основы. Если классификатор еще не обучен, ошибочный сигнал, распространяющийся по сети в процессе дообучения, окажется слишком велик, и представления, полученные на предыдущем этапе обучения, будут разрушены.

Для дообучения сети требуется выполнить следующие шаги:

1. Добавить свою сеть поверх обученной базовой сети.
2. Заморозить базовую сеть.
3. Обучить добавленную часть.



**Рис. 5.19.** Дообучение последнего сверточного блока сети VGG16

4. Разморозить несколько слоев в базовой сети.

5. Обучить эти слои и добавленную часть вместе.

Мы уже выполнили первые три шага в ходе выделения признаков. Теперь выполним шаг 4: разморозим `conv_base` и заморозим отдельные слои в ней.

Вспомним, как выглядит наша сверточная основа:

```
>>> conv_base.summary()
Layer (type)          Output Shape       Param #
=====
input_1 (InputLayer)   (None, 150, 150, 3)  0
block1_conv1 (Convolution2D) (None, 150, 150, 64) 1792
block1_conv2 (Convolution2D) (None, 150, 150, 64) 36928
block1_pool (MaxPooling2D)  (None, 75, 75, 64)  0
block2_conv1 (Convolution2D) (None, 75, 75, 128) 73856
block2_conv2 (Convolution2D) (None, 75, 75, 128) 147584
block2_pool (MaxPooling2D)  (None, 37, 37, 128)  0
block3_conv1 (Convolution2D) (None, 37, 37, 256) 295168
block3_conv2 (Convolution2D) (None, 37, 37, 256) 590080
block3_conv3 (Convolution2D) (None, 37, 37, 256) 590080
block3_pool (MaxPooling2D)  (None, 18, 18, 256)  0
block4_conv1 (Convolution2D) (None, 18, 18, 512) 1180160
block4_conv2 (Convolution2D) (None, 18, 18, 512) 2359808
block4_conv3 (Convolution2D) (None, 18, 18, 512) 2359808
block4_pool (MaxPooling2D)  (None, 9, 9, 512)   0
block5_conv1 (Convolution2D) (None, 9, 9, 512)   2359808
block5_conv2 (Convolution2D) (None, 9, 9, 512)   2359808
block5_conv3 (Convolution2D) (None, 9, 9, 512)   2359808
block5_pool (MaxPooling2D)  (None, 4, 4, 512)   0
=====
Total params: 14714688
```

Мы дообучим три последних сверточных слоя, то есть все слои выше `block4_pool` нужно заморозить, а слои `block5_conv1`, `block5_conv2` и `block5_conv3` — сделать доступными для обучения.

Почему бы не дообучить больше слоев? Почему бы не дообучить всю сверточную основу? Так можно поступить, но имейте в виду следующее.

- ❑ Начальные слои в сверточной основе кодируют более обобщенные признаки, пригодные для повторного использования, а более высокие слои кодируют более конкретные признаки. Намного полезнее донастроить более конкретные признаки, потому что именно их часто нужно перепрофилировать для решения новой задачи. Ценность дообучения нижних слоев быстро падает с их глубиной.
- ❑ Чем больше параметров обучается, тем выше риск переобучения. Сверточная основа имеет 15 миллионов параметров, поэтому было бы слишком рискованно пытаться дообучить ее целиком на нашем небольшом наборе данных.

То есть в данной ситуации лучшей стратегией будет дообучить только верхние два-три слоя сверточной основы. Сделаем это, начав с того места, на котором мы остановились в предыдущем примере.

#### **Листинг 5.22.** Замораживание всех слоев, кроме заданных

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

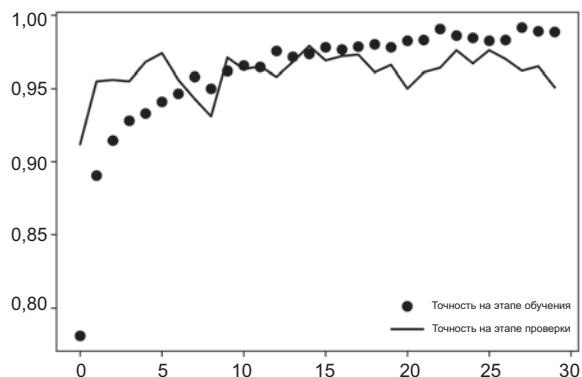
Теперь можно начинать дообучение сети. Для этого используем оптимизатор RMSProp с очень маленькой скоростью обучения. Причина использования низкой скорости обучения заключается в необходимости ограничить величину изменений, вносимых в представления трех дообучаемых слоев. Слишком большие изменения могут повредить эти представления.

#### **Листинг 5.23.** Дообучение модели

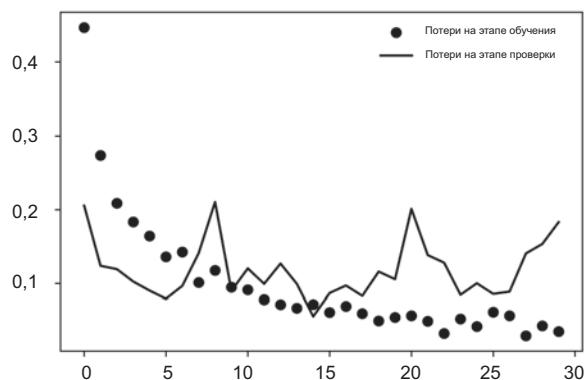
```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=50)
```

Построим графики с результатами, использовав тот же код, что и прежде (рис. 5.20 и 5.21).



**Рис. 5.20.** Точность на этапах проверки и обучения для дообучения



**Рис. 5.21.** Потери на этапах проверки и обучения для дообучения

Похоже, что кривые искажены помехами. Чтобы можно было уловить тенденцию, складим кривые, заменив фактические значения потерь и точности экспоненциальным скользящим средним. Вот простая вспомогательная функция для этого (рис. 5.22 и 5.23).

#### Листинг 5.24. Сглаживание графиков

```
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
```

```

        smoothed_points.append(point)

    return smoothed_points

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

```

Теперь кривая точности на этапе проверки выглядит более гладкой. Мы видим радующее глаз абсолютное улучшение точности на 1 %, с 96 до 97 %.

Обратите внимание, что кривая потерь не показывает реального улучшения (фактически произошло ухудшение). Вас может заинтересовать, как точность может оставаться стабильной или даже улучшаться, если потери не уменьшаются? Ответ прост: на графиках мы видим средние значения потерь по точкам; но для точности важно распределение значений потерь, а не их среднее, потому что точность есть результат определения бинарных порогов вероятности класса, предсказанного моделью. Модель может продолжать улучшаться, даже если это не отражается на среднем значении потерь.

Теперь наконец можно оценить модель на контрольных данных:

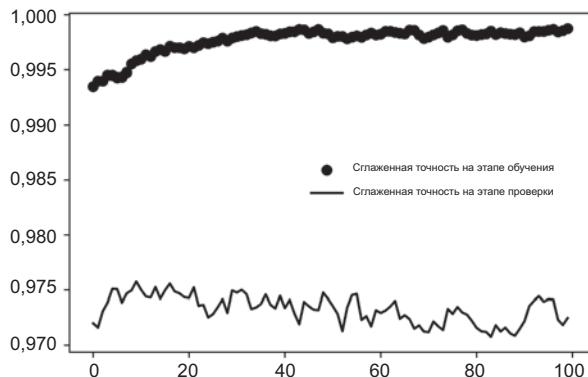
```

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary')

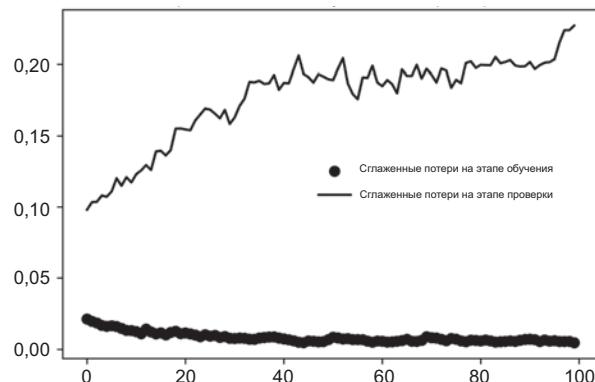
test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)

```

Здесь мы получили точность на уровне 97 %. В оригинальном состязании на сайте Kaggle, основанном на этом наборе данных, это был бы один из лучших результатов. Благодаря современным методам глубокого обучения, нам удалось достичь такого результата, использовав лишь малую часть имеющихся обучающих данных (около 10 %). Между обучением на 20 000 и на 2000 образцах огромная разница!



**Рис. 5.22.** Сглаженные кривые точности на этапах проверки и обучения для дообучения



**Рис. 5.23.** Сглаженные кривые потерь на этапах проверки и обучения для дообучения

### 5.3.3. Подведение итогов

Вот какие выводы вы должны сделать из примеров, представленных в двух предыдущих разделах:

- ❑ Сверточные нейронные сети — лучший тип моделей машинного обучения для задач распознавания образов. Вполне можно обучить такую сеть с нуля на очень небольшом наборе данных и получить приличный результат.
- ❑ Когда объем данных ограничен, главной проблемой становится переобучение. Расширение данных — эффективное средство борьбы с переобучением при работе с изображениями.
- ❑ Существующую сверточную нейронную сеть с легкостью можно повторно использовать на новом наборе данных, применив прием выделения признаков. Этот прием особенно ценен при работе с небольшими наборами изображений.
- ❑ В дополнение к выделению признаков можно использовать прием дообучения, который адаптирует к новой задаче некоторые из представлений, ранее

полученных существующей моделью. Он еще больше повышает качество модели.

Теперь у вас имеется надежный набор инструментов для решения задач классификации изображений, особенно с ограниченным объемом данных.

## 5.4. Визуализация знаний, заключенных в сверточной нейронной сети

Часто говорят, что модели глубокого обучения — это «черные ящики»: изученные ими представления сложно извлечь и представить в форме, понятной человеку. Отчасти это верно для некоторых типов моделей глубокого обучения, но уж точно не относится к сверточным нейронным сетям. Представления, изученные сверточными нейронными сетями, легко поддаются визуализации, во многом благодаря тому, что представляют собой визуальные понятия. С 2013 года был разработан широкий спектр методов визуализации и интерпретации этих представлений. Далее мы рассмотрим три наиболее доступных и практических из них:

- *визуализация промежуточных выводов сверточной нейронной сети (промежуточных активаций)* — помогает понять, как последовательность слоев сети преобразует свои входные данные, а также показывает смысл отдельных фильтров;
- *визуализация фильтров сверточной нейронной сети* — помогает точно узнать, за какой визуальный шаблон или понятие отвечает каждый фильтр;
- *визуализация тепловых карт активации класса в изображении* — помогает понять, какие части изображения идентифицируют принадлежность к заданному классу, что позволяет выявлять объекты на изображениях.

Для демонстрации первого метода — визуализации активации — мы используем небольшую сверточную нейронную сеть, обученную с нуля для классификации изображений кошек и собак в разделе 5.2. Для демонстрации двух других методов используем модель VGG16 из раздела 5.3.

### 5.4.1. Визуализация промежуточных активаций

Визуализация промежуточных активаций заключается в отображении карт признаков, которые выводятся разными сверточными и объединяющими слоями в сети в ответ на определенные входные данные (вывод слоя, результат функции активации, часто называют его *активацией*). Этот прием позволяет увидеть, как входные данные разлагаются на различные фильтры, полученные сетью в процессе обучения. Обычно для визуализации используются карты признаков с тремя измерениями: шириной, высотой и глубиной (каналы цвета). Каналы кодируют относительно независимые признаки, поэтому для визуализации этих карт признаков предпочтительнее строить двумерные изображения для каждого канала в отдельности. Начнем с загрузки модели, сохраненной в разделе 5.2:

```
>>> from keras.models import load_model
>>> model = load_model('cats_and_dogs_small_2.h5')
>>> model.summary() <1> As a reminder.
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_5 (Conv2D)	(None, 148, 148, 32)	896
<hr/>		
maxpooling2d_5 (MaxPooling2D)	(None, 74, 74, 32)	0
<hr/>		
conv2d_6 (Conv2D)	(None, 72, 72, 64)	18496
<hr/>		
maxpooling2d_6 (MaxPooling2D)	(None, 36, 36, 64)	0
<hr/>		
conv2d_7 (Conv2D)	(None, 34, 34, 128)	73856
<hr/>		
maxpooling2d_7 (MaxPooling2D)	(None, 17, 17, 128)	0
<hr/>		
conv2d_8 (Conv2D)	(None, 15, 15, 128)	147584
<hr/>		
maxpooling2d_8 (MaxPooling2D)	(None, 7, 7, 128)	0
<hr/>		
flatten_2 (Flatten)	(None, 6272)	0
<hr/>		
dropout_1 (Dropout)	(None, 6272)	0
<hr/>		
dense_3 (Dense)	(None, 512)	3211776
<hr/>		
dense_4 (Dense)	(None, 1)	513
<hr/>		
Total params:	3,453,121	
Trainable params:	3,453,121	
Non-trainable params:	0	

Далее выберем входное изображение кошки, не являющееся частью обучающего набора.

#### Листинг 5.25. Предварительная обработка единственного изображения

```
img_path = '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/
cat.1700.jpg'

from keras.preprocessing import image
import numpy as np

img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
img_tensor /= 255.

# Его форма (1, 150, 150, 3)
print(img_tensor.shape)
```

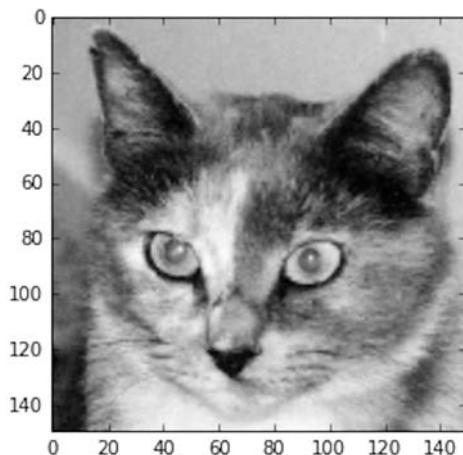
←

Преобразование изображения  
в четырехмерный тензор

←

Модель обучалась на входных  
данных, которые предварительно  
были обработаны таким способом

Отобразим исходное изображение (рис. 5.24).



**Рис. 5.24.** Тестовое изображение кошки

**Листинг 5.26.** Отображение тестового изображения

```
import matplotlib.pyplot as plt

plt.imshow(img_tensor[0])
plt.show()
```

Для извлечения карт признаков, подлежащих визуализации, создадим модель Keras, которая принимает пакеты изображений и выводит активации всех сверточных и объединяющих слоев. Для этого используем класс `Model` из фреймворка Keras. Конструктор модели принимает два аргумента: входной тензор (или список входных тензоров) и выходной тензор (или список выходных тензоров). В результате будет получен объект модели Keras, похожий на модели `Sequential`, уже знакомые вам; эта модель отображает заданные входные данные в заданные выходные данные. Отличительной чертой класса `Model` является возможность создания моделей с несколькими выходами. Более подробно класс `Model` обсуждается в разделе 7.1.

**Листинг 5.27.** Создание экземпляра модели из входного тензора и списка выходных тензоров

```
from keras import models

layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

Извлечение вывода верхних восьми слоев

Создание модели, которая вернет эти выводы с учетом заданного входа

Если передать этой модели изображение, она вернет значения активации слоев в исходной модели. Это первый пример модели с несколькими выходами в данной книге: до сих пор все представленные выше модели имели ровно один вход и один выход. Вообще модель может иметь сколько угодно входов и выходов. В частности, данная модель имеет один вход и восемь выходов: по одному на каждую активацию слоя.

#### Листинг 5.28. Запуск модели в режиме прогнозирования

```
activations = activation_model.predict(img_tensor) ←
Вернет список с пятью
 массивами NumPy: по одному
 на каждую активацию слоя
```

Возьмем для примера активацию первого сверточного слоя для входного изображения кошки:

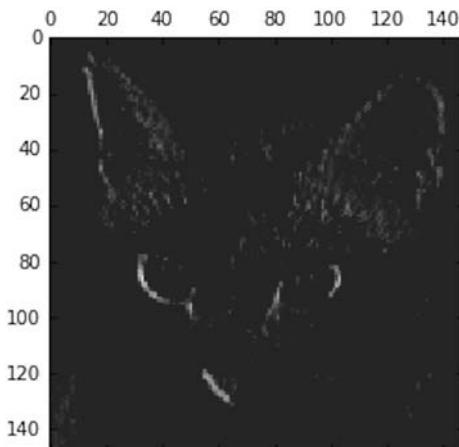
```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 148, 148, 32)
```

Это карта признаков  $148 \times 148$  с 32 каналами. Попробуем отобразить четвертый канал активации первого слоя оригинальной модели (рис. 5.25).

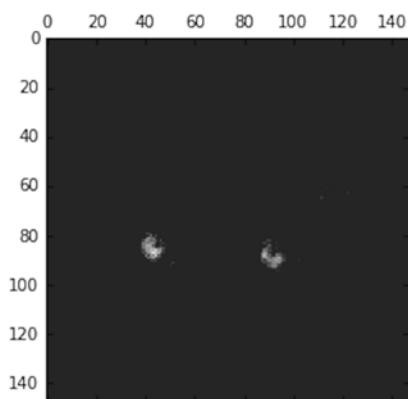
#### Листинг 5.29. Визуализация четвертого канала

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 4], cmap='viridis')
```

Похоже, что этот канал представляет собой диагональный детектор контуров. Теперь взглянем на седьмой канал (рис. 5.26) — но имейте в виду, что у вас каналы могут отличаться, потому что обучение конкретных фильтров не является детерминированной операцией.



**Рис. 5.25.** Четвертый канал активации первого слоя для тестового изображения кошки



**Рис. 5.26.** Седьмой канал активации первого слоя для тестового изображения кошки

**Листинг 5.30.** Визуализация седьмого канала

```
plt.matshow(first_layer_activation[0, :, :, 7], cmap='viridis')
```

Похоже, что это детектор «ярких зеленых точек», который может пригодиться для кодирования кошачьих глаз. Теперь построим полную визуализацию всех активаций в сети (рис. 5.27). Для этого извлечем и отобразим каждый канал во всех восьми картах активаций, поместив результаты в один большой тензор с изображениями.

**Листинг 5.31.** Визуализация всех каналов для всех промежуточных активаций

```

Карта признаков имеет форму
(1, size, size, n_features)

layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1]

    size = layer_activation.shape[1]
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0, :, :, col * images_per_row + row]
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

```

Количество признаков в карте признаков

Извлечь имена слоев для отображения на рисунке

Цикл отображения карт признаков

Количество колонок в матрице отображения каналов

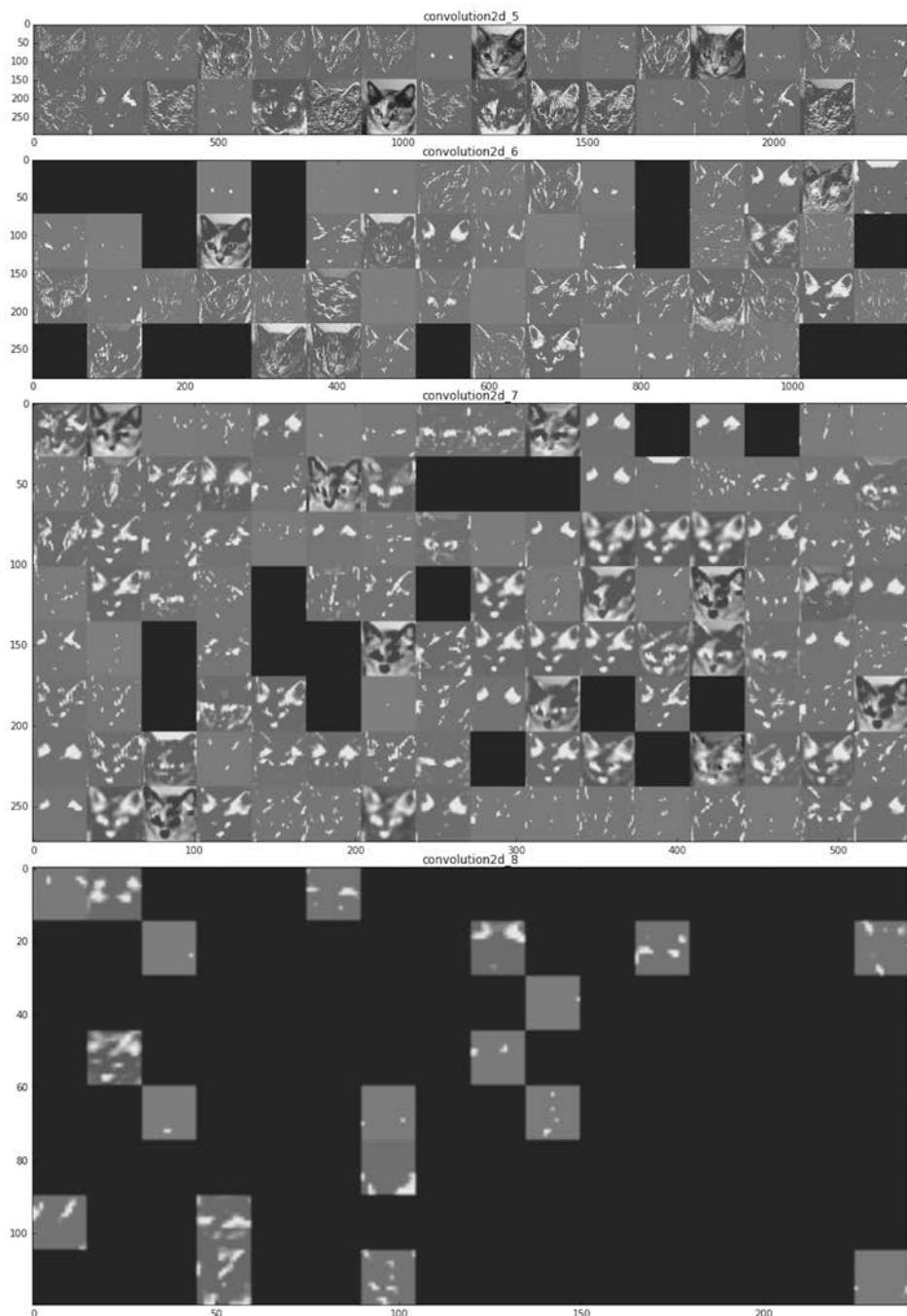
Вывод каждого фильтра в большую горизонтальную сетку

Заключительная обработка признака, чтобы получить приемлемую визуализацию

Выход сетки

Бот несколько замечаний к полученным результатам:

- ❑ Первый слой действует как коллекция разных детекторов контуров. На этом этапе активация сохраняет почти всю информацию, имеющуюся в исходном изображении.

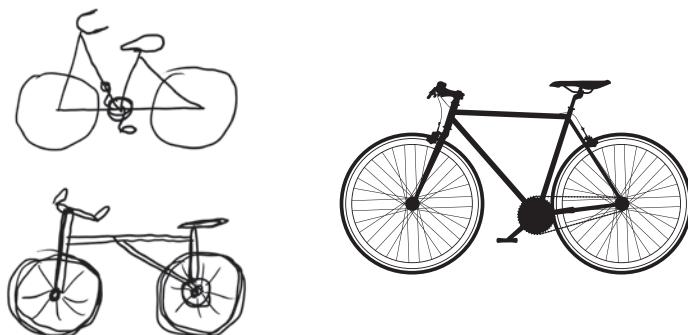


**Рис. 5.27.** Все каналы всех активаций слоев для тестового изображения кошки

- По мере подъема вверх по слоям активации становятся все более абстрактными, а их визуальная интерпретация все более сложной. Они начинают кодировать высокоуровневые понятия, такие как «кошачье ухо» или «кошачий глаз». Высокоуровневые представления несут все меньше информации об исходном изображении и все больше — о классе изображения.
- Разреженность активаций увеличивается с глубиной слоя: в первом слое все фильтры активируются исходным изображением, но в последующих слоях все больше и больше остается пустых фильтров. Это означает, что шаблон, соответствующий фильтру, не обнаруживается в исходном изображении.

Мы только что рассмотрели важную универсальную характеристику представлений, создаваемых глубокими нейронными сетями: признаки, извлекаемые слоями, становятся все более абстрактными с глубиной слоя. Активации на верхних слоях содержат все меньше и меньше информации о конкретном входном изображении и все больше и больше о цели (в данном случае о классе изображения — кошка или собака). Глубокая нейронная сеть фактически действует как *конвейер очистки информации*, который получает неочищенные исходные данные (в данном случае изображения в формате RGB) и подвергает их многократным преобразованиям, фильтруя ненужную информацию (например, конкретный внешний вид изображения) и оставляя и очищая нужную (например, класс изображения).

Примерно так же люди и животные воспринимают окружающий мир: понаблюдав сцену в течение нескольких секунд, человек запоминает, какие абстрактные объекты присутствуют в ней (велосипед, дерево), но не запоминает всех деталей внешнего вида этих объектов. Фактически при попытке нарисовать велосипед по памяти, скопив все, вам не удастся получить более или менее правильное изображение, даже при том, что вы могли видеть велосипеды тысячи раз (см. примеры на рис. 5.28). Попробуйте сделать это прямо сейчас, и вы убедитесь в справедливости сказанного. Ваш мозг научился полностью абстрагировать видимую картинку, получаемую на входе, и преобразовывать ее в высокоуровневые визуальные понятия, фильтруя при этом неважные визуальные детали и затрудняя тем самым их запоминание.



**Рис. 5.28.** Слева: попытки нарисовать велосипед по памяти.  
Справа: так должен был бы выглядеть схематичный рисунок велосипеда

## 5.4.2. Визуализация фильтров сверточных нейронных сетей

Другой простой способ исследовать фильтры, полученные сетью, — отобразить визуальный шаблон, за который отвечает каждый фильтр. Это можно сделать методом *градиентного восхождения в пространстве входов* (gradient ascent in input space): выполняя *градиентный спуск* до значения входного изображения сверточной нейронной сети, *максимизируя* отклик конкретного фильтра, начав с пустого изображения. В результате получится версия входного изображения, для которого отклик данного фильтра был бы максимальным.

Задача решается просто: нужно сконструировать функцию потерь, максимизирующую значение данного фильтра данного сверточного слоя, и затем использовать стохастический градиентный спуск для настройки значений входного изображения, чтобы максимизировать значение активации. Например, определим потери для активации фильтра 0 в уровне `block3_conv1` сети VGG16, предварительно обученной на наборе ImageNet.

**Листинг 5.32.** Определение тензора потерь для визуализации фильтра

```
from keras.applications import VGG16
from keras import backend as K

model = VGG16(weights='imagenet',
                include_top=False)

layer_name = 'block3_conv1'
filter_index = 0

layer_output = model.get_layer(layer_name).output
loss = K.mean(layer_output[:, :, :, filter_index])
```

Чтобы реализовать градиентный спуск, необходимо получить градиент потерь относительно входа модели. Для этого можно использовать функцию `gradients` из модуля `backend` фреймворка Keras.

**Листинг 5.33.** Получение градиента потерь относительно входа модели

```
grads = K.gradients(loss, model.input)[0]
```

Вызов `gradients` возвращает список тензоров (в данном случае с размером 1). Поэтому сохраняется только первый элемент (тензор)

Иногда для ускорения процесса градиентного спуска используется неочевидный трюк — нормализация градиентного тензора делением на его L2-норму (квадратный корень из усредненных квадратов значений в тензоре). Это гарантирует, что величина обновлений во входном изображении всегда будет находиться в одном диапазоне.

Теперь нужно вычислить значение тензора потерь и тензора градиента для заданного входного изображения. Для этого можно выбрать функцию из фреймворка Keras, например `iterate`, принимающую тензор Numpy (как список тензоров с единственным элементом) и возвращающую список с двумя тензорами Numpy — потерь и градиента.

#### Листинг 5.34. Трюк с нормализацией градиента

```
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5) ←
    Добавить 1e-5 перед
    операцией деления во
    избежание случайного
    деления на ноль
```

#### Листинг 5.35. Получение выходных значений Numpy для заданных входных значений Numpy

```
iterate = K.function([model.input], [loss, grads])
import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

После этого можно записать цикл, реализующий стохастический градиентный спуск.

#### Листинг 5.36. Максимизация потерь стохастическим градиентным спуском

```
Начальное изображение
с черно-белым шумом
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128. ←
    Вычисление
    значений потерь
    и градиента
step = 1. ← Величина каждого изменения градиента
for i in range(40):
    loss_value, grads_value = iterate([input_img_data]) ←
        40 шагов градиентного
        восхождения
    input_img_data += grads_value * step ←
        Корректировка входного изображения
        в направлении максимизации потерь
```

В результате получается тензор вещественных чисел с изображением, имеющий форму  $(1, 150, 150, 3)$ , значения в котором могут не быть целыми числами в диапазоне  $[0, 255]$ . Поэтому нужно в заключение превратить этот тензор в отображаемое изображение. Для этого можно воспользоваться следующей простой функцией.

#### Листинг 5.37. Функция преобразования тензора в допустимое изображение

```
def deprocess_image(x):
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1
    Нормализация: получается тензор со
    средним значением 0 и стандартным
    отклонением 0,1
    Продолжение ↗
```

**Листинг 5.37** (продолжение)

```

x += 0.5          | Ограничивает значения диапазоном [0, 1]
x = np.clip(x, 0, 1)

x *= 255         | Преобразует в массив значений RGB
x = np.clip(x, 0, 255).astype('uint8')
return x

```

Теперь у нас есть все необходимые элементы. Объединим их в функцию на Python, которая будет принимать имя слоя и индекс фильтра и возвращать тензор с допустимым изображением, представляющим собой шаблон, который максимизирует активацию заданного фильтра.

**Листинг 5.38.** Функция, генерирующая изображение, которое представляет фильтр

```

Вычисление градиента входного изображения с учетом потерь

def generate_pattern(layer_name, filter_index, size=150):
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index])

    grads = K.gradients(loss, model.input)[0]
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5) ← Трюк с нормализацией: нормализует градиент

    iterate = K.function([model.input], [loss, grads])

    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128. ←

    step = 1.
    for i in range(40):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step ← 40 шагов градиентного восхождения

    img = input_img_data[0]
    return deprocess_image(img) ← Начальное изображение с черно-белым шумом

```

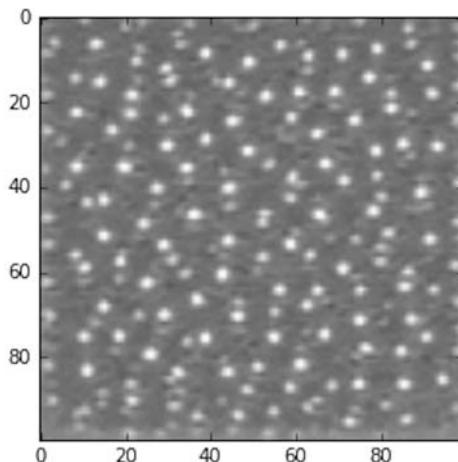
**Возврат тензоров потерь и градиента для данного входного изображения**

Взглянем на получившееся изображение (рис. 5.29):

```
>>> plt.imshow(generate_pattern('block3_conv1', 0))
```

Похоже, что фильтр с индексом 0 в слое `block3_conv1` отвечает за узор в горошек. А теперь самое интересное: мы можем визуализировать все фильтры во всех слоях. Для простоты рассмотрим только первые 64 фильтра в слое и только первые слои в каждом сверточном блоке (`block1_conv1`, `block2_conv1`, `block3_conv1`, `block4_conv1`, `block5_conv1`). Расположим получившиеся изображения шаблонов

фильтров  $64 \times 64$  в сетке  $8 \times 8$ , добавив небольшие черные границы между шаблонами (рис. 5.30–5.33).



**Рис. 5.29.** Шаблон, на который нулевой фильтр в уровне block3\_conv1 дает максимальный отклик

**Листинг 5.39.** Создание сетки со всеми шаблонами откликов фильтров в слое

```

layer_name = 'block1_conv1'                                Чистое изображение
size = 64                                                 (черный фон) для сохранения
margin = 5                                                 результатов
                                                               ←

results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin, 3)) ←

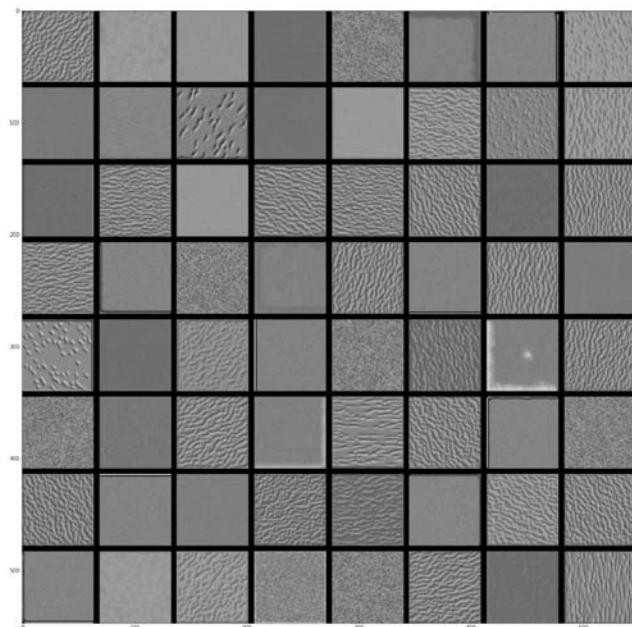
for i in range(8): ← Итерации по строкам сетки с результатами
    for j in range(8): ← Итерации по столбцам сетки с результатами
        filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

        horizontal_start = i * size + i * margin
        horizontal_end = horizontal_start + size
        vertical_start = j * size + j * margin
        vertical_end = vertical_start + size
        results[horizontal_start: horizontal_end,
vertical_start: vertical_end, :] = filter_img

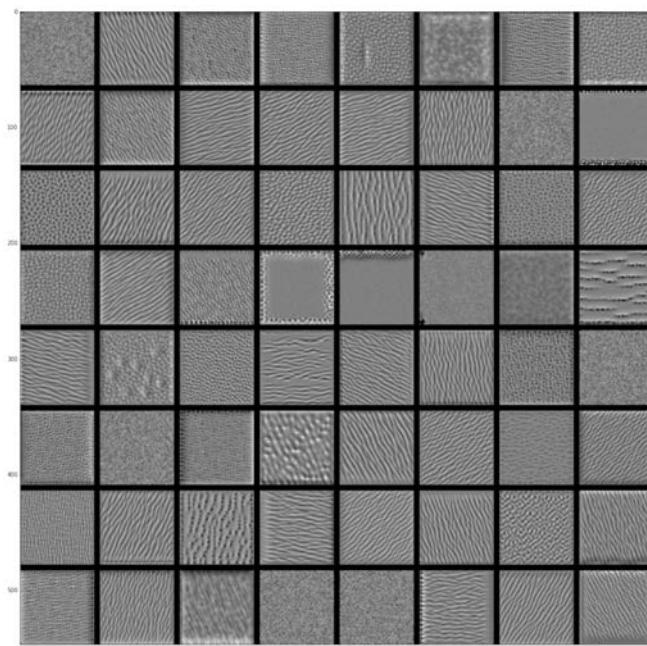
plt.figure(figsize=(20, 20)) | Отображение
plt.imshow(results)      | сетки
                                                               ←

Генерация шаблона для
фильтра  $i + (j * 8)$  в слое
с именем layer_name
  
```

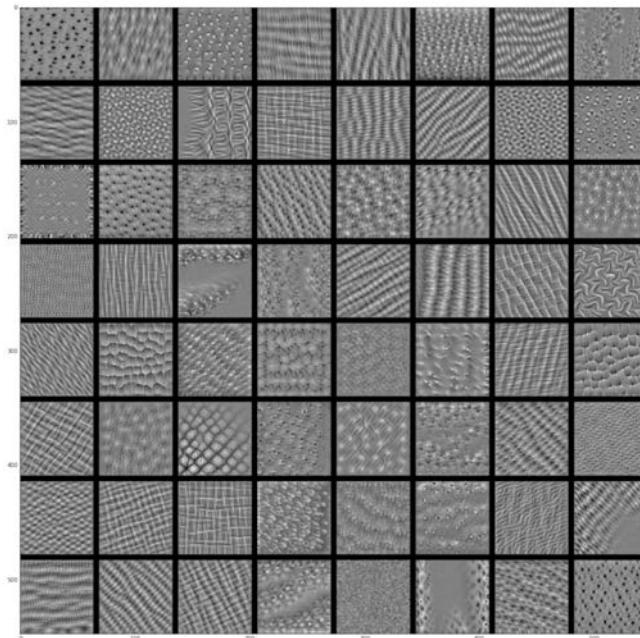
← Переписывание шаблона  
в квадрат  $(i, j)$  внутри  
сетки с результатами



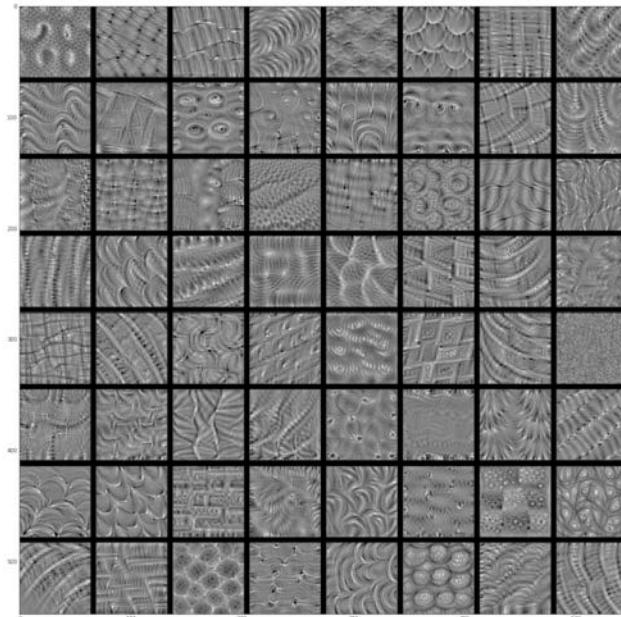
**Рис. 5.30.** Шаблоны фильтров из слоя block1\_conv1



**Рис. 5.31.** Шаблоны фильтров из слоя block2\_conv1



**Рис. 5.32.** Шаблоны фильтров из слоя block3\_conv1



**Рис. 5.33.** Шаблоны фильтров из слоя block4\_conv1

Эти визуальные представления фильтров могут многое рассказать о том, как слои сверточной нейронной сети видят мир: каждый слой в сети обучает свою коллекцию фильтров так, чтобы их входы можно было выразить в виде комбинации фильтров. Это напоминает преобразование Фурье, разлагающее сигнал в пакет косинусоидных функций. Фильтры в таких пакетах фильтров сверточной нейронной сети становятся все сложнее с увеличением слоя в модели:

- ❑ фильтры из первого слоя в модели (`block1_conv1`) кодируют простые направленные контуры и цвета (или, в некоторых случаях, цветные контуры);
- ❑ фильтры из `block2_conv1` кодируют простые текстуры, состоящие из комбинаций контуров и цветов;
- ❑ фильтры в более высоких слоях начинают напоминать текстуры, встречающиеся в естественных изображениях, — перья, глаза, листья и т. д.

### 5.4.3. Визуализация тепловых карт активации класса

В этом разделе описывается еще один прием визуализации, позволяющий понять, какие части данного изображения помогли сверточной нейронной сети принять окончательное решение о его классификации. Это полезно для отладки процесса принятия решений в сверточной нейронной сети, особенно в случае ошибок классификации. Он также помогает определить местоположение конкретных объектов на изображении.

Категория методов, описываемых здесь, называется визуализацией *карты активации класса* (Class Activation Map, CAM). Их суть заключается в создании тепловых карт активации класса для входных изображений. Тепловая карта активации класса — это двумерная сетка оценок, связанных с конкретным выходным классом и вычисляемых для каждого местоположения в любом входном изображении. Эти оценки определяют, насколько важно каждое местоположение для рассматриваемого класса. Например, для изображения, передаваемого в сверточную нейронную сеть, которая осуществляет классификацию кошек и собак, визуализация CAM позволяет генерировать тепловые карты для классов «кошка» и «собака», показывающие, насколько важными являются разные части изображения для этих классов.

Далее мы будем использовать реализацию, описанную в статье «Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization»<sup>1</sup>. Она очень проста: отобразить карту признаков для входного изображения, полученную на выходе сверточного слоя, и взвесить каждый канал в ней по градиенту класса для данного канала. Проще говоря, этот трюк заключается во взвешивании признаков в пространственной карте «как интенсивно входное изображение активирует разные каналы» по признаку «насколько важен каждый канал для данного класса». В результате получается пространственная карта признаков «как интенсивно входное изображение активирует класс».

---

<sup>1</sup> Ramprasaath R. Selvaraju et al., arXiv (2017), <https://arxiv.org/abs/1610.02391>.

Продемонстрируем этот прием с использованием предварительно обученной сети VGG16.

**Листинг 5.40.** Загрузка предварительно обученной сети VGG16

```
from keras.applications.vgg16 import VGG16
model = VGG16(weights='imagenet')
```

Обратите внимание на то, что мы добавили сверху полносвязный классификатор; во всех предыдущих случаях мы отбрасывали его

Рассмотрим фотографию двух африканских слонов на рис. 5.34 (использована на условиях лицензии Creative Commons), на которой изображены самка и ее слоненок, прогуливающиеся по саванне. Преобразуем эту фотографию в форму, которую сможет прочитать модель VGG16. Модель обучена на изображениях с размерами  $224 \times 224$ , предварительно обработанных в соответствии с правилами, реализованными в виде функции `keras.applications.vgg16.preprocess_input`, а поэтому мы также должны привести фотографию к размерам  $224 \times 224$ , преобразовать ее в тензор Numpy с числами типа `float32` и применить правила предварительной обработки.



**Рис. 5.34.** Тестовая фотография африканских слонов

**Листинг 5.41.** Предварительная обработка входного изображения для передачи в сеть VGG16

Изображение  $224 \times 224$  в формате библиотеки Python Imaging Library (PIL)

Локальный путь к целевому изображению

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input, decode_predictions
import numpy as np

img_path = '/Users/fchollet/Downloads/creative_commons_elephant.jpg'

img = image.load_img(img_path, target_size=(224, 224))
```

Продолжение ↗

**Листинг 5.41** (продолжение)

```

x = image.img_to_array(img)           ← Массив Numpy с числами типа float32,
                                     имеющий форму (224, 224, 3)
x = np.expand_dims(x, axis=0)         ← Добавление размерности для преобразования
                                     массива в пакет с формой (1, 224, 224, 3)
x = preprocess_input(x)              ← Предварительная обработка пакета
                                     (нормализация каналов цвета)

```

Теперь можно передать изображение в предварительно обученную сеть и декодировать полученный вектор в удобочитаемый формат:

```

>>> preds = model.predict(x)
>>> print('Predicted:', decode_predictions(preds, top=3)[0])
Predicted: [(u'n02504458', u'African_elephant', 0.92546833),
(u'n01871265', u'tusker', 0.070257246),
(u'n02504013', u'Indian_elephant', 0.0042589349)]

```

Вот первые три прогнозируемых класса для данного изображения:

- африканский слон (с вероятностью 92,5 %);
- кабан-секач (с вероятностью 7 %);
- индийский слон (с вероятностью 0,4 %).

Сеть распознала на изображении неопределенное количество африканских слонов. Элемент в векторе прогнозов с максимальной активацией соответствует классу «African elephant» (африканский слон) с индексом 386:

```

>>> np.argmax(preds[0])
386

```

Для визуализации части изображения, наиболее соответствующей классу «африканский слон», выполним процедуру Grad-CAM.

**Листинг 5.42.** Реализация алгоритма Grad-CAM

```

Элемент «африканский
слон» в векторе прогнозов
→ african_elephant_output = model.output[:, 386]
last_conv_layer = model.get_layer('block5_conv3') ← Выходная карта
                                                 признаков слоя block5_
                                                 conv3, последнего
                                                 сверточного слоя
                                                 в сети VGG16
Градиент класса «африканский
слон» для выходной карты
признаков слоя block5_conv3
→ grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]
                                                 Вектор с формой (512,), каждый элемент которого
                                                 определяет интенсивность градиента для заданного
                                                 канала в карте признаков
pooled_grads = K.mean(grads, axis=(0, 1, 2)) ←

```

```

iterate = K.function([model.input],
                    [pooled_grads, last_conv_layer.output[0]])

→ pooled_grads_value, conv_layer_output_value = iterate([x])

for i in range(512):
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

heatmap = np.mean(conv_layer_output_value, axis=-1) ←
Значения этих двух величин в виде
массивов NumPy для данного
образца изображения двух слонов ←
Среднее для каналов
в полученной карте
признаков — это тепловая
карта активации класса
Позволяет получить доступ к значениям
только что определенных величин: pooled_
grads и выходной карте признаков слова
block5_conv3 для заданного изображения ←
Умножает
каждый
канал в карте
признаков на
«важность этого
канала» для
класса «слон»

```

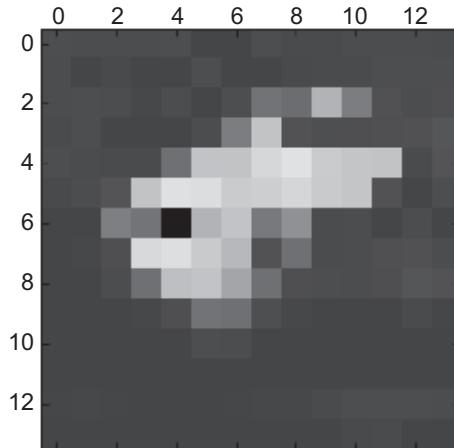
Для нужд визуализации нормализуем тепловую карту, приведя значения в ней к диапазону от 0 до 1. Результат показан на рис. 5.35.

#### Листинг 5.43. Заключительная обработка тепловой карты

```

heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)

```



**Рис. 5.35.** Тепловая карта активации класса «африканский слон» для тестового изображения

В заключение используем библиотеку OpenCV, чтобы получить фотографию слонов с наложенной на нее тепловой картой (рис. 5.36).

**Листинг 5.44.** Наложение тепловой карты на исходное изображение

```

import cv2

img = cv2.imread(img_path) ←

heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0])) ←
Изменение размеров тепловой карты в соответствии с размерами оригинальной фотографии

heatmap = np.uint8(255 * heatmap) ←
Преобразование тепловой карты в формат RGB

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET) ←
0,4 — это коэффициент интенсивности тепловой карты

superimposed_img = heatmap * 0.4 + img ←

cv2.imwrite('/Users/fchollet/Downloads/elephant_cam.jpg', superimposed_img) ←
Сохранение изображения на диск

```

Наложение тепловой карты на оригинальную фотографию



**Рис. 5.36.** Оригинальная фотография с наложенной тепловой картой активации класса

Этот прием визуализации помогает ответить на два важных вопроса:

- Почему сеть решила, что на фотографии изображен африканский слон?
- Где на фотографии находится африканский слон?

Интересно отметить, что уши слоненка оказались сильно активированы: вероятно, именно по этому признаку сеть отличает африканских слонов от индийских.

## Краткие итоги главы

- ❑ Сверточные нейронные сети — лучший инструмент для решения задач распознавания образов.
- ❑ Сверточные нейронные сети создают иерархию модульных шаблонов и понятий для представления видимого мира.
- ❑ Представления, получаемые сверточными нейронными сетями, легко поддаются исследованию, а значит, такие сети не являются черными ящиками!
- ❑ Теперь вы сможете обучать свои сверточные нейронные сети с нуля для решения своей задачи распознавания образов.
- ❑ Вы познакомились с приемом расширения данных для борьбы с переобучением.
- ❑ Вы узнали, как использовать предварительно обученные сверточные нейронные сети с применением приемов выделения признаков и дообучения.
- ❑ Вы научились визуализировать фильтры, полученные сверточной нейронной сетью, а также тепловые карты активации классов.

# 6 Глубокое обучение для текста и последовательностей

Эта глава охватывает следующие темы:

- ✓ предварительную обработку текстовых данных;
- ✓ рекуррентные нейронные сети;
- ✓ одномерные сверточные нейронные сети для обработки последовательностей

Эта глава исследует модели глубокого обучения, пригодные для обработки текста (который можно интерпретировать как последовательности слов или символов), временных последовательностей и последовательностей данных в целом. Двумя фундаментальными алгоритмами глубокого обучения для обработки последовательностей являются *рекуррентные нейронные сети* и *одномерные сверточные нейронные сети* — одномерная версия двумерных сверточных нейронных сетей, которые рассматривались в предыдущих главах. В этой главе мы обсудим оба подхода.

В число прикладных применений этих алгоритмов попадают:

- классификация документов и временных последовательностей, например, с целью идентификации темы статьи или автора книги;
- сравнение временных последовательностей, например, для оценки тесноты связи между двумя документами или двумя биржевыми котировками;
- обучение типа «последовательность в последовательность», например, для перевода последовательности английских слов на французский язык;
- анализ эмоциональной окраски, например, для классификации твитов или отзывов к фильмам на положительные и отрицательные;
- прогнозирование временных последовательностей, например, для предсказания погоды в определенном месте на основе недавних метеорологических данных.

Примеры в этой главе нацелены на решение двух конкретных задач: анализ эмоциональной окраски отзывов в наборе данных IMDB (мы уже пытались решить ее ранее) и прогноз температуры. Однако те методы, которые будут применяться для их решения, с успехом можно использовать для решения всех задач, перечисленных выше, и многих других.

## 6.1. Работа с текстовыми данными

Текст — одна из самых распространенных форм последовательностей данных. Его можно интерпретировать и как последовательность символов, и как последовательность слов, но чаще текст обрабатывается на уровне слов. Модели глубокого обучения для обработки последовательностей, представленные в следующем разделе, могут на основе текста формировать понимание естественного языка в простейшей форме, достаточной для таких применений, как классификация документов, анализ эмоциональной окраски, идентификация автора и даже получение ответов на вопросы (в ограниченном контексте). Конечно, нужно помнить, что ни одна из этих моделей в действительности не понимает текст в человеческом смысле; они лишь отражают статистическую структуру письменного языка — этого достаточно для решения многих простых задач обработки текста. Глубокое обучение для обработки естественного языка — это распознавание образов для слов, предложений и абзацев, примерно так же, как компьютерное зрение — это распознавание образов для пикселов.

Как любые другие нейронные сети, модели глубокого обучения не могут принимать на входе простой текст — они работают только с числовыми тензорами. *Векторизация* текста — это процесс преобразования текста в числовые тензоры. Ее можно выполнить несколькими способами:

- разбить текст на слова и преобразовать каждое слово в вектор;
- разбить текст на символы и преобразовать каждый символ в вектор;
- извлечь *n*-граммы из слов или символов и преобразовать каждую *n*-грамму в вектор. *N*-граммы — это перекрывающиеся группы из нескольких последовательных слов или символов.

Собирательно разные единицы, на которые можно разбить текст (слова, символы или *n*-граммы), называют *токенами*, а разбиение текста на такие токены называют *токенизацией*. Все процессы векторизации текста заключаются в применении некоторой схемы токенизации и последующем связывании числовых векторов с полученными токенами. Эти векторы упаковываются в тензоры последовательностей и передаются в нейронные сети глубокого обучения. Есть несколько способов связать вектор с токеном. В этом разделе мы познакомимся с двумя из них: *прямое кодирование токенов* и *векторное представление токенов* (обычно используется только в отношении слов и называется также *векторным представлением слов*). В оставшейся части этого раздела рассказывается об особенностях этих приемов

и демонстрируется их применение для преобразования исходного текста в тензор NumPy, который можно передать в сеть Keras.



**Рис. 6.1.** Преобразование текста в токены и затем в векторы

## N-ГРАММЫ И МЕШКИ СЛОВ

N-граммы слов — это группы по  $N$  последовательных слов, которые можно извлечь из предложения. Та же идея применима к символам.

Вот простой пример. Рассмотрим предложение «The cat sat on the mat» (кошка села на коврик). Его можно разложить на следующий набор 2-грамм<sup>1</sup>:

```
{ "The", "The cat", "cat", "cat sat", "sat",
  "sat on", "on", "on the", "the", "the mat", "mat" }
```

Также его можно разложить на такой набор 3-грамм<sup>2</sup>:

```
{ "The", "The cat", "cat", "cat sat", "The cat sat",
  "cat sat", "sat on", "on", "on the", "the",
  "sat on the", "the mat", "mat", "on the mat" }
```

Такие наборы называют *мешком биграмм* или *мешком триграмм* соответственно. Термин *мешок* в данном случае отражает тот факт, что вы имеете дело с множеством токенов, а не со списком или последовательностью: токены в мешке не упорядочены. Это семейство методов токенизации называют *мешком слов*.

Поскольку мешок слов не сохраняет порядок следования токенов (сгенерированный набор токенов интерпретируется как множество, а не как последовательность и не поддерживает общую структуру предложений), этот метод обычно используется в поверхностных моделях обработки естественного языка и крайне редко в моделях глубокого обучения. Извлечение n-грамм — еще одна форма конструирования признаков, но в глубоком обучении этот ломкий и ограниченный метод заменяют конструированием иерархических признаков. Одномерные и рекуррентные нейронные сети, которые будут представлены далее в этой главе, способны получать представления для групп слов и символов без явного определения таких групп, просматривая последовательности

<sup>1</sup> Или биграмм. — Примеч. пер.

<sup>2</sup> Или триграмм. — Примеч. пер.

слов или символов. По этой причине мы больше не будем касаться  $n$ -грамм в этой книге. Однако имейте в виду, что они являются мощным и часто обязательным инструментом конструирования признаков при использовании легковесных, поверхностных моделей обработки текста, таких как логистическая регрессия и случайные леса.

### 6.1.1. Прямое кодирование слов и символов

Прямое кодирование (one-hot encoding) — наиболее используемый и простой способ преобразования токенов в векторы. Вы уже видели его в действии в первых примерах с наборами данных IMDB и Reuters в главе 3 (где он применялся к словам). Он заключается в присваивании каждому слову уникального целочисленного индекса  $i$  с последующим его преобразованием в бинарный вектор размера  $N$  (размер словаря); все элементы этого вектора содержат нули, кроме  $i$ -го элемента, которому присваивается 1.

Конечно, прямое кодирование можно выполнить и на уровне символов. Чтобы понять, что фактически делает прямое кодирование и как оно реализуется, в листингах 6.1 и 6.2 показаны два простых примера: один для слов и другой для символов.

#### Листинг 6.1. Прямое кодирование на уровне слов (упрощенный пример)

Исходные данные: один элемент — один образец (в данном случае образцы представляют по одному предложению, но точно так же это могли бы быть целые документы)

Токенизация образцов с помощью метода `split`. В действующем приложении также желательно было бы удалить знаки пунктуации и специальные символы

```
import numpy as np

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

token_index = {} ← Создание индекса всех токенов в данных
for sample in samples:
    for word in sample.split():
        if word not in token_index:
            token_index[word] = len(token_index) + 1

max_length = 10

results = np.zeros(shape=(len(samples),
                        max_length,
                        max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1.0
```

Присваивание уникального индекса каждому уникальному слову. Обратите внимание: индекс со значением 0 не используется

Векторизация образцов. В каждом образце рассматриваются только первые  $\text{max\_length}$  слов

Здесь сохраняются результаты

**Листинг 6.2.** Прямое кодирование на уровне символов (упрощенный пример)

```
import string

samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable ← Все отображаемые символы ASCII
token_index = dict(zip(characters, range(1, len(characters) + 1)))

max_length = 50
results = np.zeros((len(samples), max_length, max(token_index.keys()) + 1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

Следует отметить, что во фреймворке Keras имеются встроенные утилиты для прямого кодирования простых текстовых данных на уровне слов и символов. Воспользуйтесь этими утилитами, поскольку они обладают рядом важных свойств, таких как удаление из строк специальных символов и возможность принимать в расчет только  $N$  слов, наиболее часто встречающихся в наборе данных (типичное ограничение, помогающее избежать создания слишком обширного пространства входных векторов).

**Листинг 6.3.** Использование Keras для прямого кодирования слов

Создание токенизатора и его настройка на учет только 1000 наиболее часто используемых слов

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(samples) ← Создание индекса всех слов
```

► sequences = tokenizer.texts\_to\_sequences(samples)

one\_hot\_results = tokenizer.texts\_to\_matrix(samples, mode='binary')

word\_index = tokenizer.word\_index ←  
print('Found %s unique tokens.' % len(word\_index))

Преобразование строк в списки  
целочисленных индексов

Так можно узнать  
вычисленный индекс слова

Прямые бинарные  
представления  
можно получить  
непосредственно.  
Этот токенизатор  
поддерживает также  
другие режимы  
векторизации

Прием прямого кодирования имеет разновидность — так называемое *прямое хеширование признаков* (one-hot hashing trick), которое можно использовать, когда словарь содержит слишком большое количество токенов, чтобы его можно было

использовать явно. Вместо явного присваивания индекса каждому слову и сохранения ссылок на эти индексы в словаре можно хешировать слова в векторы фиксированного размера. Обычно для этого используются очень легковесные функции хеширования. Главное достоинство этого метода — отсутствие необходимости хранить индексы слов, что позволяет сэкономить память и кодировать данные по мере необходимости (векторы токенов можно генерировать сразу же, по мере их обхода, до просмотра всех имеющихся данных). Единственный недостаток — этот метод восприимчив к *хеш-коллизиям*: два разных слова могут получить одинаковые хеш-значения, и впоследствии любая модель машинного обучения не сможет различить эти слова. Вероятность хеш-коллизий снижается, когда размер пространства хеширования намного больше общего количества уникальных токенов, подвергаемых хешированию.

#### **Листинг 6.4.** Прямое кодирование на уровне слов с использованием хеширования (упрощенный пример)

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

► dimensionality = 1000
max_length = 10

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = abs(hash(word)) % dimensionality ←
        results[i, j, index] = 1.
```

Слова будут сохраняться как векторы с размером 1000.

Если число слов близко к 1000 (или даже больше),

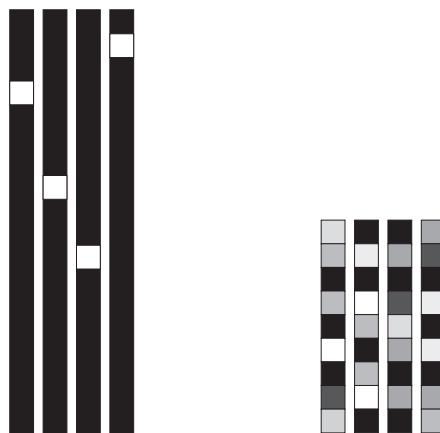
вы увидите множество хеш-коллизий, снижающих

точность этого метода кодирования

Хеширование слов  
в случайные целочисленные  
индексы между 0 и 1000

### 6.1.2. Векторное представление слов

Другим популярным и мощным способом связывания вектора со словом является использование плотных *векторов слов*, или *векторного представления слов* (word embeddings). В отличие от векторов, полученных прямым кодированием, — бинарных, разреженных (почти полностью состоящих из нулей) и с большой размерностью (их размерность совпадает с количеством слов в словаре) — векторные представления слов являются малоразмерными векторами вещественных чисел (то есть плотными векторами, в противоположность разреженным), как показано на рис. 6.2. В отличие от векторов, полученных прямым кодированием, векторные представления слов конструируются из данных. При работе с огромными словарями размерность векторов слов нередко может достигать 256, 512 или 1024. С другой стороны, прямое кодирование слов обычно влечет за собой создание векторов с числом измерений 20 000 или больше (при использовании словаря с 20 000 токенов, как в данном случае). Иначе говоря, векторное представление слов позволяет уместить больший объем информации в меньшее число измерений.



Векторы, полученные прямым кодированием:  
 - разреженные;  
 - с большим числом размерностей;  
 - негибкие

Векторные представления:  
 - плотные;  
 - малоразмерные;  
 - конструируются на основе данных

**Рис. 6.2.** Представления слов, полученные прямым кодированием или хешированием, являются разреженными, негибкими и имеют большое число размерностей, тогда как векторные представления — плотные, относительно малоразмерные и конструируются на основе данных

Получить векторные представления слов можно двумя способами:

- ❑ Конструировать векторные представления в процессе решения основной задачи (такой, как классификация документа или определение эмоциональной окраски). В этом случае изначально создаются случайные векторы слов, которые затем постепенно конструируются (обучаются), как это происходит с весами нейронной сети.
- ❑ Загрузить в модель векторные представления, полученные с использованием другой задачи машинного обучения, отличной от решаемой. Такие представления называют *предварительно обученными векторными представлениями слов*.

Рассмотрим оба способа.

### Конструирование векторных представлений слов с помощью слоя Embedding

Простейший способ связать плотный вектор со словом — выбрать случайный вектор. Однако пространство векторов, которое получится в этом случае, не имеет структуры: например, слова *accurate* и *exact* могут в конечном счете получить совершенно разные векторные представления, даже при том, что в большинстве

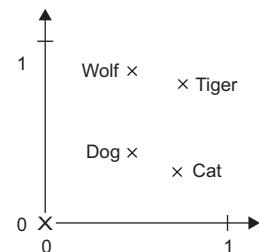
случаев они взаимозаменяемы<sup>1</sup>. Глубокой нейронной сети трудно будет понять такое искаженное, неструктурированное пространство векторов.

Говоря более абстрактно, геометрические отношения между векторами слов должны отражать семантические связи между соответствующими им словами. Как предполагается, векторные представления слов должны отображать человеческий язык в геометрическое пространство. Например, от правильно сконструированного пространства векторных представлений разумно ожидать, что синонимы будут представлены похожими векторами и в целом геометрическое расстояние ( $L_2$ -расстояние) между любыми двумя векторами будет зависеть от семантического расстояния между соответствующими словами (слова с далеким друг от друга смыслом будут представлены далекими друг от друга точками, а слова со схожим смыслом — близкими). Кроме расстояния, может оказаться желательным наделить определенным смыслом конкретные *направления* в пространстве векторов. Поясним это на конкретном примере.

На рис. 6.3 изображена двумерная плоскость с четырьмя векторными представлениями слов: *кошка*, *собака*, *волк* и *тигр*. С выбранными здесь векторными представлениями некоторые семантические отношения между словами можно выразить в виде геометрических преобразований. Например, один и тот же вектор позволяет перейти от *кошки* к *тигру* и от *собаки* к *волку*: этот вектор можно было бы интерпретировать как вектор «от домашнего животного к дикому». Аналогично, другой вектор позволяет перейти от *собаки* к *кошке* и от *волка* к *тигру*, и его можно интерпретировать как вектор «от псовых к кошачьим».

В настоящих векторных пространствах слов типичными примерами осмыслиенных геометрических преобразований могут служить векторы «половая принадлежность» и «много». Например, сложив векторы «женщина» и «король», мы получили бы вектор «королева». Сложив векторы «много» и «король», мы получили бы вектор «короли». В векторных пространствах слов обычно существуют тысячи таких интерпретируемых и потенциально полезных векторов.

Существует ли идеальное векторное пространство слов, точно отражающее человеческий язык, которое можно было бы использовать для решения любых задач обработки естественного языка? Возможно, однако нам еще предстоит вычислить нечто подобное. Кроме того, нет такого понятия, как *человеческий язык*, — есть много разных языков, и они не изоморфны, потому что каждый язык является отражением конкретной культуры и контекста. Пригодность векторного пространства слов для практического применения в значительной степени зависит от конкретной задачи: идеальное векторное пространство слов для англоязычной модели анали-



**Рис. 6.3.** Упрощенный пример векторного пространства слов

<sup>1</sup> Оба слова — *accurate* и *exact* — в большинстве контекстов переводятся одинаково, как точный, верный, четкий, пунктуальный. — Примеч. пер.

за эмоциональной окраски отзывов к фильмам может отличаться от идеального векторного пространства для англоязычной модели классификации юридических документов, потому что важность определенных семантических отношений различна для разных задач.

Как следствие, представляется разумным обучать новое векторное пространство слов для каждой новой задачи. К счастью, прием обратного распространения ошибки помогает легко добиться этого, а Keras еще больше упрощает реализацию. Речь идет об обучении весов слоя: в данном случае слоя **Embedding**.

#### Листинг 6.5. Создание слоя Embedding

```
from keras.layers import Embedding  
embedding_layer = Embedding(1000, 64) ←
```

Слой **Embedding** принимает как минимум два аргумента: количество возможных токенов (в данном случае 1000: 1 + максимальный индекс слова) и размерность пространства (в данном случае 64)

Слой **Embedding** лучше всего воспринимать как словарь, отображающий целочисленные индексы (обозначающие конкретные слова) в плотные векторы. Он принимает целые числа на входе, отыскивает их во внутреннем словаре и возвращает соответствующие векторы. Это эффективная операция поиска в словаре (рис. 6.4).

Индекс слова → Слой Embedding → Вектор, соответствующий слову

**Рис. 6.4. Слой Embedding**

Слой **Embedding** получает на входе двумерный тензор с целыми числами и с формой (**образцы**, **длина\_последовательности**), каждый элемент которого является последовательностью целых чисел. Он может работать с последовательностями разной длины: например, слою **Embedding** из предыдущего примера можно передавать пакеты с формой (32, 10) (пакет с 32 последовательностями, каждая длиной 10) или (64, 15) (пакет с 64 последовательностями, каждая длиной 15). Все последовательности в пакете должны иметь одинаковую длину, потому что упаковываются в один тензор, поэтому короткие последовательности, если они есть, нужно дополнить нулями, а длинные — усечь.

Этот слой возвращает трехмерный тензор с вещественными числами и с формой (**образцы**, **длина\_последовательности**, **размерность\_векторного\_представления**). Такой трехмерный тензор можно затем обработать слоем RNN или одномерным сверточным слоем (оба будут представлены в следующих разделах).

При создании слоя **Embedding**, его веса (внутренний словарь векторов токенов) инициализируются случайными значениями, как в случае с любым другим слоем. В процессе обучения векторы слов постепенно корректируются посредством обратного распространения ошибки, и пространство превращается в структурированную модель, пригодную к использованию. После полного обучения пространство векторов приобретет законченную структуру, специализированную под решение конкретной задачи.

Используем эту идею для решения задачи определения эмоциональной окраски отзывов к фильмам в IMDB, которую мы уже пытались решить ранее. Сначала подготовим исходные данные. Ограничимся набором из 10 000 слов, наиболее часто встречающихся в отзывах к фильмам (так же, как мы делали это в первый раз), и выберем из каждого отзыва только первые 20 слов. Сеть будет обучать 8-мерные векторные представления, по 10 000 слов в каждом, преобразовывать входные последовательности целых чисел (двумерный тензор с целыми числами) в векторные последовательности (трехмерный тензор с вещественными числами), преобразовывать трехмерный тензор в двумерный и обучать единственный слой Dense на вершине, предназначенный для классификации.

#### Листинг 6.6. Загрузка данных из IMDB для передачи в слой Embedding

```
from keras.datasets import imdb
from keras import preprocessing

max_features = 10000
maxlen = 20
```

Количество слов, рассматриваемых как признаки

Обрезка текста после этого количества слов (в числе max\_features самых распространенных слов)

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features)
```

Загрузка данных как списков целых чисел

```
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

Преобразование списков целых чисел в двумерный тензор с целыми числами и с формой (образцы, максимальная\_длина)

#### Листинг 6.7. Использование слоя Embedding и классификатора данных из IMDB

Определение максимальной длины входа для слоя Embedding в целях последующего уменьшения размерности. После слоя Embedding активация имеет форму (образцы, максимальная\_длина, 8)

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding

model = Sequential()
model.add(Embedding(10000, 8, input_length=maxlen))

model.add(Flatten())
```

Преобразование трехмерного тензора векторов в двумерный тензор с формой (образцы, максимальная\_длина \* 8)

```
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['acc'])
model.summary()
```

```
history = model.fit(x_train, y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.2)
```

Добавление классификатора сверху

На проверочных данных мы получили точность ~76%. Это очень хорошо, если учесть, что мы исследовали только первые 20 слов из каждого отзыва. Однако обратите внимание на то, что в результате простого сокращения размерности векторных последовательностей и обучения единственного слоя `Dense` получается модель, которая отдельно интерпретирует каждое слово во входной последовательности, не учитывая связей между словами и структуры предложений (например, эта модель наверняка расценит оба отзыва — «*this movie is a bomb*» и «*this movie is the bomb*» — как отрицательные<sup>1</sup>). Лучший результат можно получить, если добавить рекуррентные или одномерные сверточные слои поверх векторных последовательностей для извлечения признаков, которые учитывают целые последовательности слов. Об этом мы поговорим в следующих нескольких разделах.

## Использование предварительно обученных векторных представлений слов

Иногда обучающих данных оказывается слишком мало, чтобы можно было обучить векторное представление слов для конкретной задачи. Что можно сделать в этом случае?

Вместо обучения векторного представления совместно с решением задачи можно загрузить предварительно сформированные векторные представления, хорошо организованные и обладающие полезными свойствами, которые охватывают основные аспекты языковой структуры. Использование предварительно обученных векторных представлений слов в обработке естественного языка обосновывается почти так же, как использование предварительно обученных сверточных нейронных сетей в классификации изображений: отсутствием достаточного объема данных для выделения по-настоящему мощных признаков. Также предполагается, что для решения задачи достаточно обобщенных признаков, то есть обобщенных визуальных и семантических признаков. В данном случае есть смысл повторно использовать признаки, выделенные в ходе решения другой задачи.

Такие векторные представления обычно вычисляются с использованием статистик встречаемости слов (наблюдений совместной встречаемости слов в предложениях или документах) и применением разнообразных методик, иногда с привлечением нейронных сетей, иногда нет. Идея плотных, малоразмерных пространств векторных представлений слов, обучаемых без учителя, первонациально была исследована Йошуа Бенгио (Yoshua Bengio) с коллегами в начале 2000-х годов<sup>2</sup>, но более основательное ее исследование и практическое применение началось только после выхода одной из самых известных и успешных схем реализации векторного представления слов — алгоритма Word2vec (<https://code>.

---

<sup>1</sup> Словосочетание «*a bomb*» носит отрицательный оттенок в отношении кино, а словосочетание «*the bomb*», напротив, выражает восхищение. Фраза «*this movie is a bomb*» переводится как «этот фильм — полный провал», а «*this movie is the bomb*» как «этот фильм — бомба!» в нашем привычном понимании. — Примеч. пер.

<sup>2</sup> Yoshua Bengio et al., Neural Probabilistic Language Models (Springer, 2003).

[google.com/archive/p/word2vec](http://google.com/archive/p/word2vec)), разработанного в 2013 году Томасом Миколовым (Tomas Mikolov) из компании Google. Измерения Word2vec охватывают такие семантические признаки, как пол.

Существует множество разнообразных предварительно обученных векторных представлений слов, которые можно загрузить и использовать в слое `Embedding`. Word2vec — одно из них. Другое популярное представление называется «глобальные векторы представления слов» (Global Vectors for Word Representation, GloVe, <https://nlp.stanford.edu/projects/glove>) и разработано исследователями из Стэнфорда в 2014-м. Это представление основано на факторизации матрицы статистик совместной встречаемости слов. Его создатели включили в представление миллионы токенов из английского языка, полученных из Википедии и данных компании Common Crawl.

Давайте посмотрим, как можно использовать представления GloVe в моделях Keras. Ту же методику можно применить к Word2vec и другим предварительно обученным векторным представлениям слов. Этот пример также можно использовать для усовершенствования приемов токенизации текста, представленных несколькими абзацами выше: вы начинаете с простого текста и постепенно движетесь вверх.

### 6.1.3. Объединение всего вместе: от исходного текста к векторному представлению слов

Мы будем использовать модель, похожую на ту, по которой только что прошли: преобразуем предложения в последовательности векторов, снизим их размерность и обучим слой `Dense` сверху. Но за основу мы возьмем предварительно обученные векторные представления слов и вместо использования предварительно токенизованных данных IMDB, входящих в состав Keras, пройдем весь процесс с самого начала — с загрузки исходных текстовых данных.

#### Загрузка данных из IMDB в виде простого текста

Сначала загрузите архив с исходным набором данных IMDB, доступный по адресу <http://mng.bz/0tIo>. Распакуйте его.

Теперь соберем отдельные обучающие отзывы в список строк, по одной строке на отзыв. Также соберем метки отзывов (положительный/отрицательный) в список `labels`.

#### Листинг 6.8. Обработка меток из исходного набора данных IMDB

```
import os

imdb_dir = '/Users/fchollet/Downloads/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')
```

Продолжение ↗

**Листинг 6.8** (продолжение)

```
labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

## Токенизация данных

Теперь преобразуем текст в векторное представление и подготовим обучающую и проверочную выборки, использовав идеи, представленные выше в этом разделе. Поскольку предварительно обученные векторные представления слов особенно полезны в случаях, когда доступны лишь небольшие объемы обучающих данных (в иных ситуациях лучше создавать представления, специализированные для конкретных задач, которые могут оказаться более качественными), мы добавим следующий трюк: ограничим набор обучающих данных первыми 200 образцами. Другими словами, мы попытаемся получить модель классификации отзывов, обучив ее всего на 200 примерах.

**Листинг 6.9.** Токенизация текста из исходного набора данных IMDB

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100 ← Отсечение остатка отзывов после 100-го слова
training_samples = 200 ← Обучение на выборке из 200 образцов
validation_samples = 10000 ← Проверка на выборке из 10 000 образцов
max_words = 10000 ← Рассмотрение только 10 000 наиболее часто используемых слов

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)
```

```

indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]

```

← Разбивка данных на обучающую и проверочную выборки, но перед этим данные перемешиваются, поскольку отзывы в исходном наборе упорядочены (сначала следуют отрицательные, а потом положительные)

## Загрузка векторного представления GloVe

Откройте в браузере страницу <https://nlp.stanford.edu/projects/glove> и загрузите векторные представления, предварительно обученные на данных из англоязычной Википедии за 2014 год. Этот ZIP-архив размером 822 Мбайт с именем *glove.6B.zip* содержит 100-мерные векторы с 400 000 слов (токенов). Распакуйте его.

### Предварительная обработка векторных представлений

Обработаем распакованный файл *.txt* и создадим индекс, отображающий слова (в виде строк) в их векторные представления (в виде векторов с числами).

#### Листинг 6.10. Обработка файла с векторными представлениями слов GloVe

```

glove_dir = '/Users/fchollet/Downloads/glove.6B'
embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))

```

Теперь создадим матрицу векторных представлений, которую можно будет передать на вход слоя *Embedding*. Это должна быть матрица с формой (**максимальное\_число\_слов**, **размерность\_представления**), каждый элемент *i* которой содержит вектор с размером, равным размерности представления, соответствующий слову с индексом *i* в индексе (созданном в ходе токенизации). Обратите внимание: индекс 0 не должен соответствовать никакому слову или токену — это пустой элемент.

#### Листинг 6.11. Подготовка матрицы векторных представлений слов GloVe

```

embedding_dim = 100
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

```

Словам, отсутствующим в индексе представлений, будут соответствовать векторы с нулевыми значениями

## Определение модели

Используем модель с той же архитектурой, как было показано выше.

### Листинг 6.12. Определение модели

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

## Загрузка представлений GloVe в модель

Уровень `Embedding` имеет единственную весовую матрицу: двумерную матрицу с вещественными числами, каждый  $i$ -й элемент которой — это вектор, связанный с  $i$ -м словом в индексе. Все довольно просто. Загрузим подготовленную матрицу GloVe в слой `Embedding` — первый слой модели.

### Листинг 6.13. Загрузка предварительно обученных векторных представлений слов в слой Embedding

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

Мы также заморозили слой `Embedding` (присвоив атрибуту `trainable` значение `False`). Причины те же, что были описаны, когда мы знакомились с особенностями применения предварительно обученных сверточных нейронных сетей: когда в модели имеются уже обученные части (как наш слой `Embedding`) и части, инициализированные случайными значениями (как наш классификатор), обученные части не должны изменяться в ходе обучения, чтобы не потерять свои знания. Большие изменения градиента, вызванные случайными начальными значениями в необученных слоях, могут оказать разрушительное влияние на обученные слои.

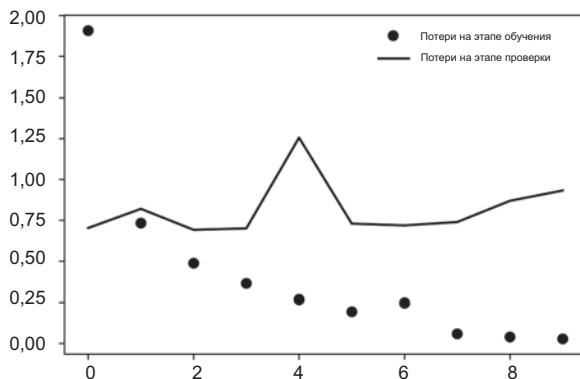
## Обучение и оценка модели

Скомпилируем и обучим модель.

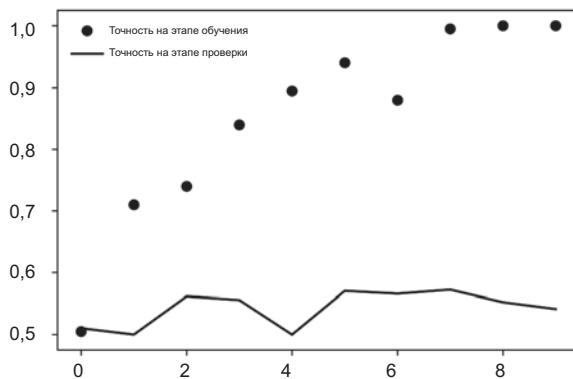
### Листинг 6.14. Обучение и оценка

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                     epochs=10,
                     batch_size=32,
                     validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

Теперь выведем графики изменения качества модели (рис. 6.5 и 6.6).



**Рис. 6.5.** Потери на этапах обучения и проверки при использовании уже обученных векторных представлений слов



**Рис. 6.6.** Точность на этапах обучения и проверки при использовании уже обученных векторных представлений слов

### Листинг 6.15. Вывод результатов

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
```

Продолжение ↗

**Листинг 6.15** (продолжение)

```
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Модель быстро достигает состояния переобучения, что неудивительно при таком малом объеме обучающих данных. По этой причине оценка точности демонстрирует высокую изменчивость, но все же достигает уровня 50 %.

Имейте в виду, что ваши результаты могут несколько отличаться от представленных, поскольку при таком небольшом объеме обучающих данных результаты сильно зависят от того, какие именно 200 образцов попадут в обучающую выборку при случайном выборе. Если вы получили худший результат, чем мы, попробуйте ради эксперимента отобрать другой набор из 200 случайных образцов (в реальной жизни у вас не будет такой возможности).

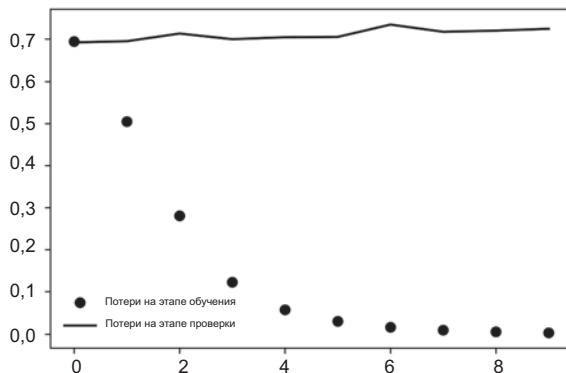
Эту же модель можно обучить без загрузки предварительно обученных векторных представлений слов и без замораживания слоя `Embedding`. В этом случае будет обучено представление, узкоспециализированное для входного набора токенов. В такой ситуации обычно получается более мощное представление, чем предварительно обученное, если имеется большой объем обучающих данных. Но в нашем случае имеется всего 200 обучающих образцов. Тем не менее давайте попробуем проделать это (рис. 6.7 и 6.8).

**Листинг 6.16.** Обучение той же модели без использования уже обученных векторных представлений

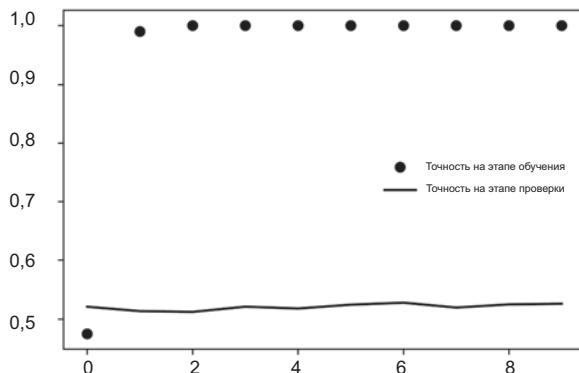
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=32,
                      validation_data=(x_val, y_val))
```



**Рис. 6.7.** Потери на этапах обучения и проверки без использования уже обученных векторных представлений слов



**Рис. 6.8.** Точность на этапах обучения и проверки без использования уже обученных векторных представлений слов

Точность на этапе проверки замерла на уровне, близком к 50 %. То есть в данном случае предварительно обученные векторные представления слов выигрывают у вновь обученных. Если увеличить число обучающих образцов, ситуация быстро изменится на противоположную, — попробуйте ради эксперимента.

Наконец, оценим модель на контрольной выборке. Сначала токенизируем контрольные данные.

#### Листинг 6.17. Токенизация данных из контрольной выборки

```
test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
```

Продолжение ↗

**Листинг 6.17** (продолжение)

```
dir_name = os.path.join(test_dir, label_type)
for fname in sorted(os.listdir(dir_name)):
    if fname[-4:] == '.txt':
        f = open(os.path.join(dir_name, fname))
        texts.append(f.read())
        f.close()
    if label_type == 'neg':
        labels.append(0)
    else:
        labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

А затем загрузим и оценим первую модель.

**Листинг 6.18.** Оценка модели на контрольном наборе данных

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

Мы получили удручающе низкую точность 56 %. Сложно добиться хорошего результата, имея лишь горстку обучающих образцов!

### 6.1.4. Подведение итогов

Теперь вы умеете:

- преобразовывать текст в нечто, что можно передать в нейронную сеть;
- использовать слой `Embedding` в модели Keras для обучения специализированных векторных представлений токенов;
- использовать предварительно обученные векторные представления слов для получения дополнительных выгод в небольших задачах обработки естественного языка.

## 6.2. Рекуррентные нейронные сети

Главной характеристикой всех нейронных сетей, с которыми мы познакомились к данному моменту, таких как полносвязные и сверточные нейронные сети, является отсутствие памяти. Каждый вход обрабатывается ими независимо, без сохранения состояния между ними. Чтобы с помощью таких сетей обработать последовательность или временной ряд данных, необходимо передать в сеть всю последовательность целиком, преобразовав ее в единый пакет. Именно так мы поступили в предыдущем примере: мы преобразовали все отзывы из IMDB в один большой вектор и обработали его целиком. Такие сети называют *сетями прямого распространения* (*feedforward networks*).

С другой стороны, читая предложение, мы осмысливаем его слово за словом, быстро перескакивая глазами с одного на другое и запоминая предыдущие; это позволяет нам постепенно вникать в смысл, передаваемый предложением. Биологический интеллект воспринимает информацию последовательно, сохраняя внутреннюю модель обрабатываемого, основываясь на предыдущей информации и постоянно дополняя эту модель по мере поступления новой информации.

*Рекуррентная нейронная сеть* (Recurrent Neural Network, RNN) использует тот же принцип, хотя и в чрезвычайно упрощенном виде: она обрабатывает последовательность, перебирая ее элементы и сохраняя *состояние*, полученное при обработке предыдущих элементов. Фактически RNN — это разновидность нейронной сети, имеющей внутренний цикл (рис. 6.9). Сеть RNN сбрасывает состояние между обработкой двух разных, независимых последовательностей (таких, как два разных отзыва из IMDB), поэтому одна последовательность все еще интерпретируется как единый блок данных: единственный входной пакет. Однако теперь блок данных обрабатывается не за один шаг; сеть выполняет внутренний цикл, перебирая последовательность элементов.

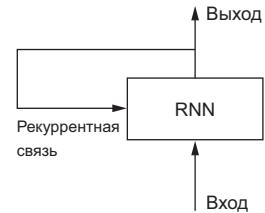
Чтобы пояснить понятия «цикл» и «состояние», реализуем средствами NumPy простую сеть RNN с прямой передачей. Она будет принимать на входе последовательность векторов в виде двумерного тензора с формой (*временные\_интервалы*, *входные\_признаки*), перебирать временные интервалы и, учитывая текущее состояние и входные признаки (с формой (*входные\_признаки*,)) в момент  $t$ , конструировать выходной результат, соответствующий моменту  $t$ . Этот результат затем будет сохраняться во внутреннем состоянии как подготовка к следующей итерации. Для первого временного интервала предыдущий выходной результат не определен; в этот момент сеть не имеет текущего состояния. Поэтому текущее состояние первоначально инициализируется вектором с нулевыми значениями элементов, который называют *начальным состоянием* сети.

Ниже представлена реализация этой RNN в псевдокоде.

#### Листинг 6.19. Реализация RNN в псевдокоде

```
state_t = 0 ← Состояние в момент t
for input_t in input_sequence: ← Цикл по последовательности элементов
    output_t = f(input_t, state_t)
    state_t = output_t ← Предыдущее выходное значение становится текущим состоянием для следующей итерации
```

Функцию  $f$  можно конкретизировать еще больше: она преобразует входные данные и состояние в выходной результат и параметризуется двумя матрицами,  $W$  и  $U$ , и вектором смещений. Она напоминает полносвязный слой в сети прямого распространения.



**Рис. 6.9.** Рекуррентная сеть — сеть с циклом

**Листинг 6.20.** Более подробная реализация RNN в псевдокоде

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

Чтобы сделать эти понятия абсолютно однозначными, напишем упрощенную реализацию сети RNN на основе Numpy.

**Листинг 6.21.** Реализация сети RNN на основе Numpy

```

Начальное состояние: вектор
с нулевыми значениями элементов

import numpy as np

timesteps = 100 ← Число временных интервалов во входной последовательности
input_features = 32 ← Размерность пространства входных признаков
output_features = 64 ← Размерность пространства выходных признаков

inputs = np.random.random((timesteps, input_features)) ←

► state_t = np.zeros((output_features,))

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,)) | Создание матриц
со случайными весами

successive_outputs = []
for input_t in inputs: ← input_t — вектор с формой (входные_признаки,)
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t) | Сохранение выходных
                                            данных в список

    state_t = output_t ←

final_output_sequence = np.concatenate(successive_outputs, axis=0) ←

Объединение входных данных
с текущим состоянием (выходными
данными на предыдущем шаге)

Входные данные: случайный
шум для простоты примера

Окончательный результат — двумер-
ный тензор с формой (временные_ин-
тервалы, выходные_признаки)

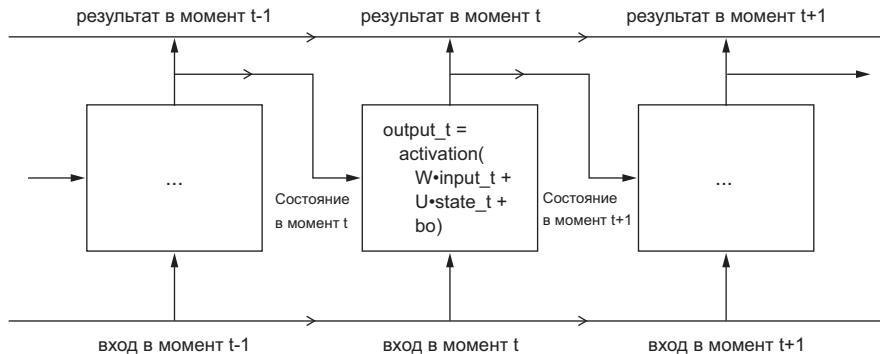
Обновление текущего состояния
сети как подготовка к обработке
следующего временного интервала

```

Довольно просто: как видите, RNN — это цикл `for`, который повторно использует величины, вычисленные в предыдущей итерации, и не более того. Конечно, вы могли бы сконструировать множество разных сетей RNN, соответствующих этому определению, и этот пример — одна из простейших реализаций RNN. Рекуррентные

сети характеризуются функцией, реализующей один шаг, такой как следующая, использованная в данном примере (рис. 6.10):

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```



**Рис. 6.10.** Простая рекуррентная сеть, развернутая во времени

### ПРИМЕЧАНИЕ

В этом примере конечный результат имеет вид двумерного тензора с формой (`временные_интервалы, выходные_признаки`), где каждый временной интервал — это результат цикла в момент времени  $t$ . Каждому временному интервалу  $t$  в выходном тензоре соответствует информация о временных интервалах от  $0$  до  $t$  во входной последовательности — обо всем прошлом. Поэтому во многих случаях нет необходимости иметь всю последовательность результатов; достаточно получить последний результат (значение `output_t` по окончании цикла), так как он уже содержит информацию обо всей последовательности.

## 6.2.1. Рекуррентный слой в Keras

Процессу, который мы только что реализовали с применением Numpy, соответствует фактический слой в Keras — слой `SimpleRNN`:

```
from keras.layers import SimpleRNN
```

С одним незначительным отличием: `SimpleRNN` обрабатывает пакеты последовательностей, как и все другие слои в Keras, а не единственную последовательность, как наш предыдущий пример. Это означает, что он принимает входные данные с формой (`размер_пакета, временные_интервалы, входные_признаки`), а не (`временные_интервалы, входные_признаки`).

Подобно всем рекуррентным слоям в Keras, `SimpleRNN` может действовать в двух разных режимах: возвращать полные последовательности результатов для всех временных интервалов (трехмерный тензор с формой (`размер_пакета, временные_интервалы, выходные_признаки`)) или же возвращать только последний результат (одномерный вектор с формой (`размер_пакета, выходные_признаки`)).

ные\_интервалы, выходные\_признаки)) или только последний результат для каждой входной последовательности (двумерный тензор с формой (размер\_пакета, входные\_признаки)). Выбор режима управляется аргументом `return_sequences` конструктора. Рассмотрим пример, в котором используется слой `SimpleRNN` и возвращается результат только для последнего временного интервала:

```
>>> from keras.models import Sequential
>>> from keras.layers import Embedding, SimpleRNN
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32))
>>> model.summary()

Layer (type) Output Shape Param #
=====
embedding_22 (Embedding) (None, None, 32) 320000
=====
simperrnn_10 (SimpleRNN) (None, 32) 2080
=====
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

Следующий пример возвращает полную последовательность состояний:

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()

Layer (type) Output Shape Param #
=====
embedding_23 (Embedding) (None, None, 32) 320000
=====
simperrnn_11 (SimpleRNN) (None, None, 32) 2080
=====
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

Иногда полезно наложить друг на друга несколько рекуррентных слоев, чтобы увеличить репрезентативность сети. В таких ситуациях все промежуточные слои должны возвращать полные последовательности результатов:

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32)) ← Последний слой возвращает только последний результат
```

```
>>> model.summary()

Layer (type) Output Shape Param #
=====
embedding_24 (Embedding)      (None, None, 32)    320000
simplernn_12 (SimpleRNN)     (None, None, 32)    2080
simplernn_13 (SimpleRNN)     (None, None, 32)    2080
simplernn_14 (SimpleRNN)     (None, None, 32)    2080
simplernn_15 (SimpleRNN)     (None, 32)          2080
=====
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

Теперь попробуем применить такую же модель для решения задачи классификации отзывов к фильмам из набора данных IMDB. Сначала подготовим данные.

### Листинг 6.22. Подготовка данных IMDB

```
from keras.datasets import imdb
from keras.preprocessing import sequence
max_features = 10000
maxlen = 500
batch_size = 32
print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(
    num_words=max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)
```

Обучим простую рекуррентную сеть, состоящую из слоев Embedding и SimpleRNN.

### Листинг 6.23. Обучение модели со слоями Embedding и SimpleRNN

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN, Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
```

Продолжение ↗

**Листинг 6.23** (продолжение)

```
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)
```

Теперь выведем графики изменения величины потерь и точности модели на этапах обучения и проверки (рис. 6.11 и 6.12).

**Листинг 6.24.** Вывод результатов

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

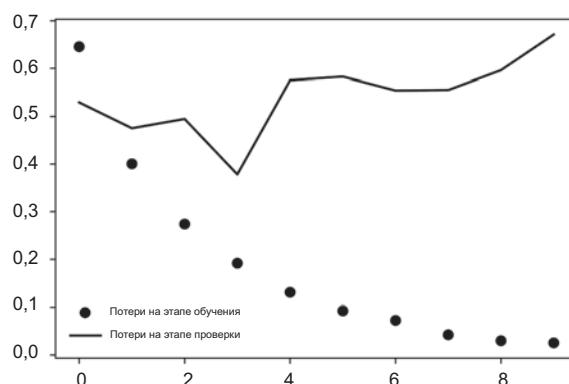
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

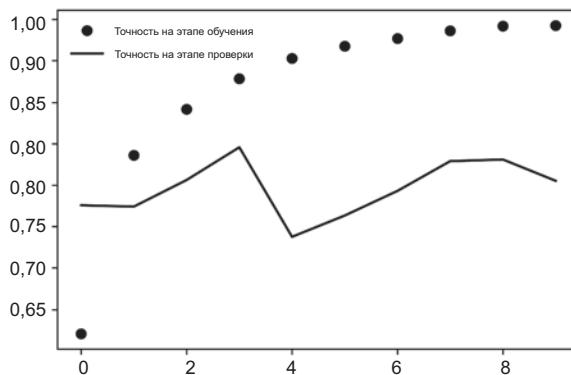
plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



**Рис. 6.11.** Потери на этапах обучения и проверки для модели анализа IMDB со слоем SimpleRNN



**Рис. 6.12.** Точность на этапах обучения и проверки для модели анализа IMDB со слоем SimpleRNN

Как вы помните, первое упрощенное решение этой задачи в главе 3 показало точность 88 % на контрольных данных. К сожалению, эта маленькая рекуррентная сеть не смогла достичь того же уровня (показав на этапе проверки точность лишь 85 %). Отчасти проблема объясняется тем, что анализу подвергаются только 500 первых слов в каждом отзыве, а не вся последовательность, следовательно, сеть RNN получает на входе меньше информации, чем модель, с которой мы ее сравниваем. Другая причина в том, что слой SimpleRNN плохо подходит для обработки длинных последовательностей, таких как текст.

Другие типы рекуррентных слоев позволяют добиться более высоких результатов. Рассмотрим несколько таких более продвинутых слоев.

## 6.2.2. Слои LSTM и GRU

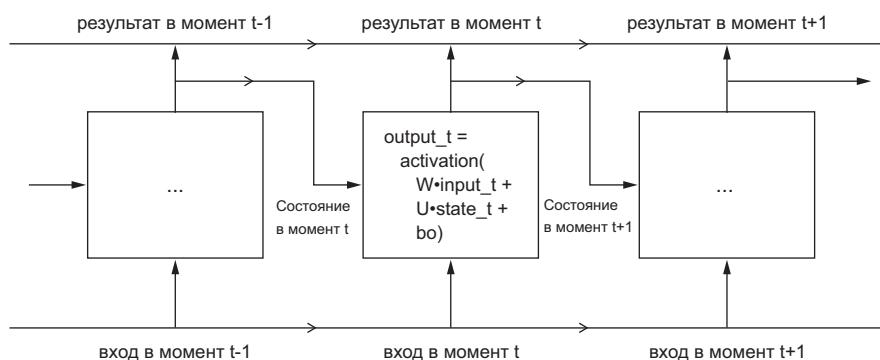
SimpleRNN не единственный рекуррентный слой, доступный в Keras. Кроме него имеются также слои LSTM и GRU. В практических решениях они используются чаще, потому что SimpleRNN слишком прост для реальных задач. SimpleRNN страдает одной существенной проблемой: теоретически в каждый момент времени  $t$  он должен хранить информацию о входных данных за многочисленные предыдущие интервалы времени, но на практике такие протяженные зависимости не поддаются обучению. Это связано с *проблемой затухания градиента*, напоминающего эффект, который наблюдается в нерекуррентных сетях (сетях прямого распространения) с большим количеством слоев: по мере увеличения количества слоев сеть в конечном итоге становится необучаемой. Теоретическое обоснование этого эффекта было дано Хохрейтером (Hochreiter), Шмидхубером (Schmidhuber) и Бенгио (Bengio) в начале 1990-х<sup>1</sup>. Слои LSTM и GRU создавались специально для решения этой проблемы.

<sup>1</sup> См., например: Yoshua Bengio, Patrice Simard, and Paolo Frasconi, «Learning Long-Term Dependencies with Gradient Descent Is Difficult», *IEEE Transactions on Neural Networks* 5, no. 2 (1994).

Рассмотрим слой LSTM. Лежащий в его основе алгоритм долгой краткосрочной памяти (Long Short-Term Memory, LSTM) был разработан Хохрейтером и Шмидхубером в 1997<sup>1</sup>; он стал кульминацией их исследований проблемы затухания градиента.

Этот слой является вариантом слоя SimpleRNN, уже знакомого вам; он добавляет поддержку переноса информации через многие интервалы времени. Вообразите конвейерную ленту, движущуюся параллельно обрабатываемой последовательности. Информация из последовательности может в любой момент перекладываться на конвейерную ленту, переноситься к более поздним интервалам времени и сниматься с ленты, если она необходима. В этом заключается суть работы слоя LSTM: он сохраняет информацию для последующего использования, тем самым предотвращая постепенное затухание старых сигналов во время обработки.

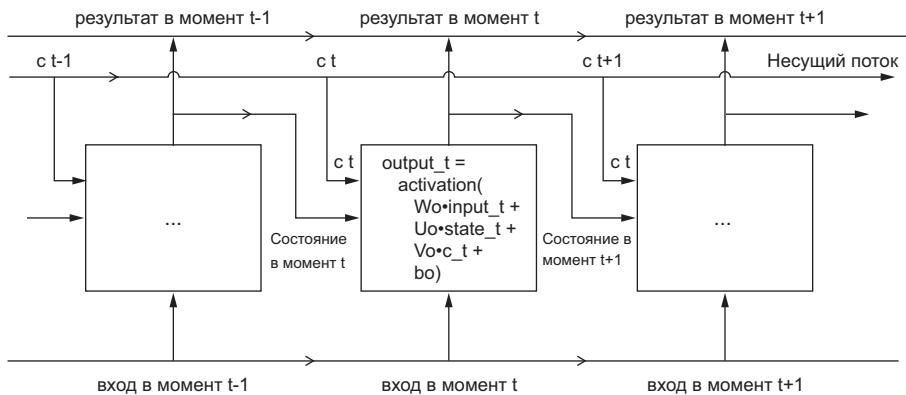
Чтобы разобраться более детально, начнем с ячейки SimpleRNN (рис. 6.13). Так как у нас имеется большое количество весовых матриц, выходные матрицы  $W$  и  $U$  в ячейке мы обозначим индексом  $o$  ( $W_o$  и  $U_o$ ) — от англ. *output* (выходной, на выходе).



**Рис. 6.13.** Начальная точка слоя LSTM: слой SimpleRNN

Добавим в эту схему дополнительный поток данных, несущий информацию сквозь интервалы времени. Для его значений в разные интервалы времени будем использовать обозначение  $C_t$ , где  $C$  — от англ. *carry* (перенесенный). Эта информация будет оказывать следующее влияние на ячейку: объединяться с входящей и рекуррентной связями (путем плотного преобразования: скалярное произведение на весовую матрицу с добавлением смещения и применением функции активации) и влиять на состояние, передаваемое в следующий интервал времени (через функцию активации и операцию умножения). Концептуально, поток переноса информации осуществляет модулирование следующего результата и следующего состояния (рис. 6.14). Пока все довольно просто.

<sup>1</sup> Sepp Hochreiter and Jürgen Schmidhuber, «Long Short-Term Memory», *Neural Computation* 9, no. 8 (1997).



**Рис. 6.14.** Переход от SimpleRNN к LSTM: добавление несущего потока

А теперь о деталях способа вычисления следующего значения в несущем потоке данных: он основывается на трех разных преобразованиях, все три имеют форму ячеек SimpleRNN:

$$y = \text{activation}(\text{dot}(\text{state}_t, U) + \text{dot}(\text{input}_t, W) + b)$$

Однако эти преобразования имеют свои весовые матрицы, которые мы обозначим индексами  $i$ ,  $f$  и  $k$ . Вот что мы имеем (это может показаться необоснованным, но наберитесь терпения, я все объясню позже).

#### Листинг 6.25. Реализация архитектуры LSTM в псевдокоде (1/2)

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(C_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

Получим новое переносимое состояние (далее обозначается как  $c_t$ ), объединив  $i_t$ ,  $f_t$  и  $k_t$ .

#### Листинг 6.26. Реализация архитектуры LSTM в псевдокоде (1/2)

```
c_t+1 = i_t * k_t + c_t * f_t
```

Добавим это в общую картину, как показано на рис. 6.15. Вот и все. Совсем несложно, просто немного замысловато.

Если хотите удариться в философию, подумайте о том, что делает каждая из этих операций. Например, можно сказать, что умножение  $c_t$  на  $f_t$  — это способ преднамеренного забывания ненужной информации в несущем потоке данных. А умножение  $i_t$  на  $k_t$  представляет собой информацию о настоящем, добавляя новую информацию в несущий поток. Но в конечном счете эти интерпретации не имеют

большого значения, потому что *фактическое* действие операций определяется содержимым параметризующих их весов, а веса вычисляются непрерывно и заново в каждом цикле обучения, что делает невозможным приписать какую-то конкретную цель той или иной операции. Спецификация ячейки RNN (как только что было описано) определяет ваше пространство гипотез — пространство, в котором в процессе обучения вы будете искать оптимальные настройки модели, однако она не определяет, что именно делает ячейка, — это зависит от весов в ячейке. Одна и та же ячейка с разными весами может действовать совершенно иначе. Поэтому набор операций, образующих ячейку RNN, лучше рассматривать как набор ограничений в вашем поиске, а не как *дизайн* в инженерном смысле.

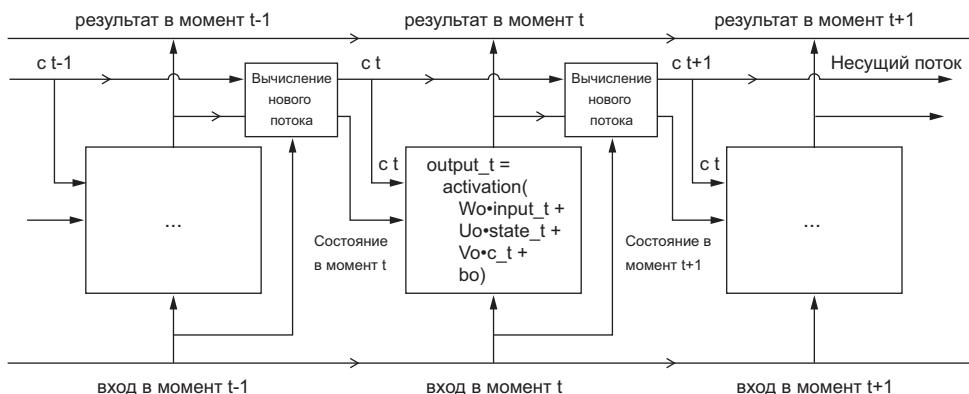


Рис. 6.15. Анатомия LSTM

С точки зрения исследователя, выбор таких ограничений — особенностей реализации ячеек RNN — лучше переложить на алгоритмы оптимизации (такие, как обобщенные алгоритмы обучения с подкреплением) и избавить людей-инженеров от него. В будущем именно так мы и будем строить сети. Подводя итог, можно сказать, что от вас не требуется понимания особенности архитектуры ячейки LSTM; это не ваша задача как человека. Просто помните назначение ячейки LSTM: позволить прошлой информации повторно внедриться в процесс обучения и оказать сопротивление проблеме затухания градиента.

### 6.2.3. Пример использования слоя LSTM из Keras

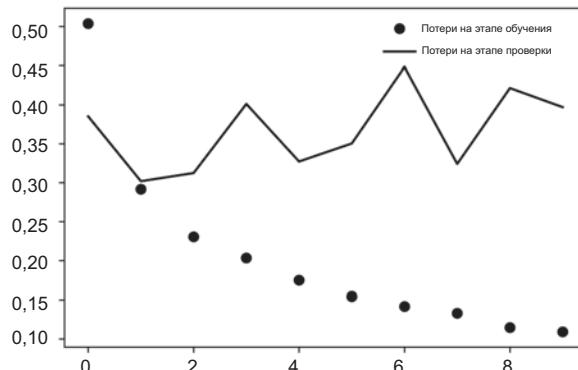
Теперь перейдем от теории к практике: создадим модель со слоем LSTM и обучим ее на данных IMDB (рис. 6.16 и 6.17). Новая сеть похожа на предыдущую, со слоем SimpleRNN. Мы указали только размерность результата слоя LSTM, оставив другие аргументы (а их довольно много) со значениями по умолчанию. Значения по умолчанию подобраны в Keras очень хорошо и пригодны для большинства ситуаций, что избавляет нас от необходимости тратить время на настройку параметров вручную.

**Листинг 6.27.** Использование слоя LSTM из Keras

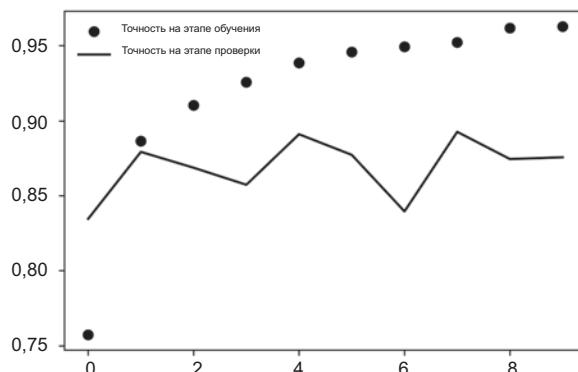
```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)
```



**Рис. 6.16.** Потери на этапах обучения и проверки для модели анализа IMDB со слоем LSTM



**Рис. 6.17.** Точность на этапах обучения и проверки для модели анализа IMDB со слоем LSTM

На этот раз мы достигли точности 89 % на этапе проверки. Неплохой результат: намного лучше, чем сетью на основе слоя SimpleRNN, что в значительной степени объясняется меньшей подверженностью LSTM проблеме затухания градиента, — и немного лучше, чем при использовании полносвязного решения в главе 3, даже при том, что в этом примере объем обучающих данных был меньше, чем в главе 3. Здесь мы ограничиваем последовательности 500 словами, тогда как в главе 3 в обучении участвовали полные последовательности.

Однако этот результат не является впечатляющим для подхода с таким большим объемом вычислений. Почему решение на основе LSTM не смогло добиться лучшего результата? Одна из причин — мы даже не пытались настроить гиперпараметры, такие как размерность векторных представлений или размерность результата, возвращаемого слоем LSTM. Другой причиной может быть отсутствие регуляризации. Однако, если быть честными, главная причина в том, что анализ глобальной протяженной структуры отзывов (с чем прекрасно справляется LSTM) плохо помогает в решении задачи определения эмоциональной окраски. Такие простые задачи хорошо решаются путем определения частот слов, которые встречаются в отзывах. Именно на этом было основано первое полносвязное решение. Однако существуют другие, намного более сложные задачи обработки естественного языка, где мощь LSTM проявляется более очевидно: например, в диалоговых системах типа «вопрос/ответ» и в машинном переводе.

### 6.2.4. Подведение итогов

Теперь вы знаете:

- ❑ что такое рекуррентные нейронные сети (RNN) и как они работают;
- ❑ что такое LSTM и почему на длинных последовательностях этот подход дает лучшие результаты, чем простое решение на основе RNN;
- ❑ как использовать слои RNN в Keras для обработки последовательных данных.

Далее мы рассмотрим некоторые дополнительные возможности рекуррентных сетей, которые помогут вам извлечь максимальную выгоду из последовательных моделей глубокого обучения.

## 6.3. Улучшенные методы использования рекуррентных нейронных сетей

В этом разделе мы рассмотрим три улучшенных приема повышения качества и степени обобщения рекуррентных нейронных сетей. К концу раздела вы будете знать большую часть из того, что нужно знать об использовании рекуррентных сетей в Keras. Мы продемонстрируем все три идеи на примере решения задачи прогнозирования температуры на следующие сутки, опираясь на временные по-

следовательности данных, поступающих от датчиков на крыше здания, таких как температура, атмосферное давление и влажность. Это довольно трудная задача, иллюстрирующая многие сложности, возникающие при работе с временными последовательностями.

Мы рассмотрим следующие приемы:

- ❑ *рекуррентное прореживание* — особый встроенный способ использования прореживания для борьбы с переобучением в рекуррентных слоях;
- ❑ *наложение рекуррентных слоев* — способ увеличения репрезентативности сети (за счет увеличения объема вычислений);
- ❑ *двунаправленные рекуррентные слои* — представляют одну и ту же информацию в рекуррентной сети разными способами, повышая точность и ослабляя проблемы, связанные с забыванием.

### 6.3.1. Задача прогнозирования температуры

До сих пор мы рассматривали единственную разновидность последовательных данных — текстовые данные, такие как в наборах данных IMDB и Reuters. Однако последовательные данные можно найти во многих других областях, не только в обработке естественного языка. Во всех примерах в этом разделе будут использоваться временные последовательности данных о погоде, записанных на гидрометеорологической станции в институте биогеохимии Макса Планка в Йене (Jena), Германия<sup>1</sup>.

В этот набор данных включены замеры 14 разных характеристик (таких, как температура, атмосферное давление, влажность, направление ветра и т. д.), выполнившиеся каждые 10 минут в течение нескольких лет. Вообще сбор данных был начат в 2003-м, но в этот пример включены только данные за 2009–2016 годы. Этот набор данных идеально подходит для изучения принципов работы с числовыми временными рядами. Мы воспользуемся ими для создания модели, принимающей на входе некоторые данные о погоде в ближайшем прошлом (за несколько дней) и прогнозирующей температуру воздуха на ближайшие 24 часа в будущем.

Загрузите и распакуйте архив с данными, как показано ниже:

```
cd ~/Downloads  
mkdir jena_climate  
cd jena_climate  
wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip  
unzip jena_climate_2009_2016.csv.zip
```

Посмотрим, что у нас имеется.

<sup>1</sup> Олаф Колле (Olaf Kolle), [www.bgc-jena.mpg.de/wetter](http://www.bgc-jena.mpg.de/wetter).

**Листинг 6.28.** Обзор набора метеорологических данных Jena

```
import os

data_dir = '/users/fchollet/Downloads/jena_climate'
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')

f = open(fname)
data = f.read()
f.close()

lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]

print(header)
print(len(lines))
```

Этот код выведет 420 551 строку с данными (каждая строка соответствует одному замеру и содержит дату замера и 14 значений разных параметров, имеющих отношение к погоде), а также следующий заголовок:

```
["Date Time",
 "p (mbar)",
 "T (degC)",
 "Tpot (K)",
 "Tdew (degC)",
 "rh (%)",
 "VPmax (mbar)",
 "VPact (mbar)",
 "VPdef (mbar)",
 "sh (g/kg)",
 "H2OC (mmol/mol)",
 "rho (g/m**3)",
 "wv (m/s)",
 "max. wv (m/s)",
 "wd (deg)"]
```

Теперь преобразуем все 420 551 строку с данными в массив NumPy.

**Листинг 6.29.** Преобразование данных

```
import numpy as np

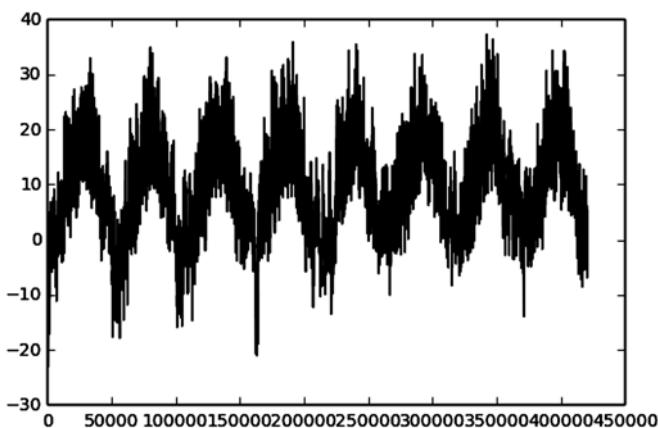
float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')][1:]
    float_data[i, :] = values
```

Для примера построим график (рис. 6.18) изменения температуры (в градусах Цельсия). На этом графике ясно виден годовой цикл изменения температуры.

**Листинг 6.30.** Создание графика изменения температуры

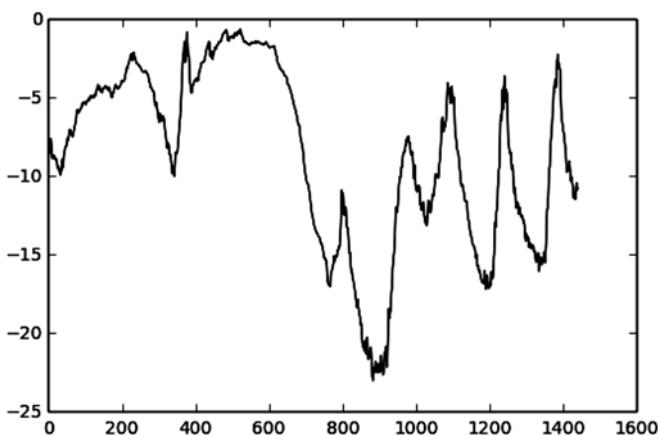
```
from matplotlib import pyplot as plt

temp = float_data[:, 1] # температура (в градусах Цельсия)
plt.plot(range(len(temp)), temp)
```



**Рис. 6.18.** График изменения температуры ( $^{\circ}\text{C}$ ), построенный по полному набору данных

Вот более короткий график изменения температуры — за первые 10 дней (рис. 6.19). Так как данные записываются каждые 10 минут, за сутки накапливается 144 замера.



**Рис. 6.19.** График изменения температуры ( $^{\circ}\text{C}$ ) по данным за первые 10 дней

**Листинг 6.31.** Создание графика изменения температуры по данным за первые 10 дней

```
plt.plot(range(1440), temp[:1440])
```

На этом графике можно видеть суточный цикл изменения температуры, который особенно четко наблюдается на отрезке, соответствующем последним четырем дням. Также отметьте, что этот 10-дневный отрезок соответствует довольно холодному зимнему месяцу.

Если бы мы предсказывали среднюю температуру на следующий месяц по данным за несколько предыдущих месяцев, эта не составило бы большого труда благодаря устойчивой периодичности в масштабах года. Однако изменение температуры в масштабе нескольких дней выглядит более хаотичным. Можно ли с высокой надежностью предсказать временную последовательность в масштабе суток? Давайте посмотрим.

### 6.3.2. Подготовка данных

Вот точная формулировка задачи: по `lookback` интервалам (один интервал равен 10 минутам) за прошлый период, из которых отобраны образцы через каждые `step` интервалов, предсказать температуру на следующие `delay` интервалов. Мы будем использовать следующие значения параметров:

- ❑ `lookback = 720` — количество наблюдений за предыдущие 5 дней;
- ❑ `step = 6` — шаг отбора образцов из наблюдений, то есть один образец за каждый час;
- ❑ `delay = 144` — целью являются следующие 24 часа в будущем.

Прежде всего нам понадобится:

- ❑ Преобразовать данные в формат, понятный нейронной сети. Это легко: данные уже представлены в числовом виде, поэтому нам не придется как-то векторизовать их. Однако все временные последовательности в данных имеют разный масштаб (например, температура обычно находится в диапазоне между  $-20$  и  $+30$ , а атмосферное давление, измеряемое в миллибарах, изменяется около значения 1000). Мы должны нормализовать временные последовательности независимо друг от друга, чтобы все они состояли из небольших по величине значений примерно одинакового масштаба.
- ❑ Написать на Python генератор, который принимает текущий массив данных и возвращает пакеты данных, представляющие собой недавнее прошлое, а также целевую температуру в будущем. Поскольку образцы в наборе данных излишне избыточны (образцы  $N$  и  $N + 1$  будут иметь много общего), было бы расточительством явно выделять и использовать каждый образец. Вместо этого мы будем генерировать образцы на лету, используя исходные данные.

В процессе обработки данных мы будем вычитать среднее для каждой временной последовательности и делить на стандартное отклонение. Для обучения мы используем первые 200 000 замеров, поэтому среднее и стандартное отклонение должны вычисляться только по этой выборке.

#### **Листинг 6.32.** Нормализация данных

```
mean = float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

В листинге 6.33 показана нужная нам функция-генератор данных. Она возвращает кортеж (*образцы, цели*), где *образцы* — это один пакет входных данных, а *цели* — соответствующий массив целевых температур. Функция принимает следующие аргументы:

- ❑ **data** — исходный массив вещественных чисел, нормализованных кодом в листинге 6.32;
- ❑ **lookback** — количество интервалов в прошлом от заданного момента, за которое отбираются входные данные;
- ❑ **delay** — количество интервалов в будущем от заданного момента, за которое отбираются целевые данные;
- ❑ **min\_index** и **max\_index** — индексы в массиве **data**, ограничивающие область для извлечения данных; это помогает оставить в неприкосновенности сегменты проверочных и контрольных данных;
- ❑ **shuffle** — флаг, определяющий порядок извлечения образцов: с перемешиванием или в хронологическом порядке;
- ❑ **batch\_size** — количество образцов в пакете;
- ❑ **step** — период в интервалах, из которого извлекается один образец; мы установим его равным 6, чтобы получить по одному образцу за каждый час.

#### **Листинг 6.33.** Функция-генератор, возвращающая временные последовательности образцов и их целей

```
def generator(data, lookback, delay, min_index, max_index,
              shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
```

Продолжение ↗

**Листинг 6.33** (продолжение)

```
i += len(rows)

samples = np.zeros((len(rows),
                   lookback // step,
                   data.shape[-1]))
targets = np.zeros((len(rows),))
for j, row in enumerate(rows):
    indices = range(rows[j] - lookback, rows[j], step)
    samples[j] = data[indices]
    targets[j] = data[rows[j] + delay][1]
yield samples, targets
```

Теперь используем абстрактную функцию-генератор для создания трех других функций-генераторов: для получения обучающих, проверочных и контрольных данных. Все они будут отбирать образцы из разных временных сегментов оригинальных данных: обучающие данные будут извлекаться из первых 200 000 интервалов, проверочные — из следующих 100 000, а контрольные — из остальных.

**Листинг 6.34.** Функции-генераторы, возвращающие обучающие, проверочные и контрольные данные

```
lookback = 1440
step = 6
delay = 144
batch_size = 128

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step,
                      batch_size=batch_size)
val_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=200001,
                     max_index=300000,
                     step=step,
                     batch_size=batch_size)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)

val_steps = (300000 - 200001 - lookback) // batch_size ←
test_steps = (len(float_data) - 300001 - lookback) // batch_size ←
```

Сколько раз нужно обратиться к val\_gen, чтобы получить проверочный набор данных целиком

Сколько раз нужно обратиться к test\_gen, чтобы получить контрольный набор данных целиком

### 6.3.3. Базовое решение без привлечения машинного обучения

Прежде чем начать использовать черные ящики моделей глубокого обучения для решения задачи прогнозирования температуры, опробуем более простой и очевидный подход. Он поможет провести базовую линию, которую мы должны будем превзойти, чтобы доказать преимущество более сложных моделей машинного обучения. Такие очевидные базовые решения могут использоваться, когда вы подступаетесь к новой задаче, не имеющей (пока) известного решения. Классическим примером могут служить несбалансированные задачи классификации, когда некоторые классы могут быть намного более распространены, чем другие. Если набор данных содержит 90 % экземпляров класса «А» и 10 % экземпляров класса «Б», тогда очевидным решением задачи классификации является неизменный выбор класса «А» для предсказания классов новых образцов. Такой классификатор будет иметь общую точность 90 %, и, соответственно, любое решение на основе машинного обучения должно превзойти эти 90 %, чтобы доказать свою полезность. Иногда такие элементарные базовые решения на удивление трудно превзойти.

В данном случае временные последовательности можно с полной уверенностью считать монотонными (температура завтра, вероятно, будет близка к сегодняшней), а также подчиняющимися суточной периодичности. То есть разумным базовым решением предсказания температуры через 24 часа является текущая температура. Давайте оценим этот подход, используя метрику средней абсолютной ошибки (Mean Absolute Error, MAE):

```
np.mean(np.abs(preds - targets))
```

Вот цикл оценки.

**Листинг 6.35.** Оценка базового решения MAE

```
def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print(np.mean(batch_maes))

evaluate_naive_method()
```

Этот цикл вернул оценку MAE, равную 0,29. Так как значения температуры нормализованы и теперь имеют среднее значение 0 и стандартное отклонение 1, данное число нельзя интерпретировать непосредственно. Средняя абсолютная ошибка в данном случае соответствует  $0,29 \times \text{temperature\_std}$  градусам Цельсия: 2,57 °C.

**Листинг 6.36.** Преобразование оценки MAE в градусы Цельсия

```
celsius_mae = 0.29 * std[1]
```

Это довольно большая средняя абсолютная ошибка. Теперь попробуем использовать наши знания в области глубокого обучения, чтобы улучшить эту оценку.

### 6.3.4. Базовое решение с привлечением машинного обучения

Перед попыткой создать такую сложную и затратную (в вычислительном смысле) модель, как рекуррентная нейронная сеть, помимо базового решения без привлечения машинного обучения также полезно попробовать найти простые и незатратные модели машинного обучения (например, неглубокую полносвязную сеть). Это лучший способ убедиться, что любые усложнения, направленные на решение задачи, оправданы и действительно дают преимущества.

В следующем листинге демонстрируется полносвязная модель, которая сначала снижает размерность данных, а затем пропускает их через два слоя `Dense`. Обратите внимание на отсутствие функции активации в последнем слое `Dense`, что характерно для задачи регрессии. В роли оценки потерь используется средняя абсолютная ошибка (MAE). Поскольку мы оцениваем те же самые данные с той же метрикой, что и в предыдущем базовом решении, их результаты можно сравнивать непосредственно.

#### Листинг 6.37. Обучение и оценка полносвязной модели

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step, float_data.
shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Выведем кривые потерь на этапах обучения и проверки (рис. 6.20).

#### Листинг 6.38. Вывод результатов

```
import matplotlib.pyplot as plt

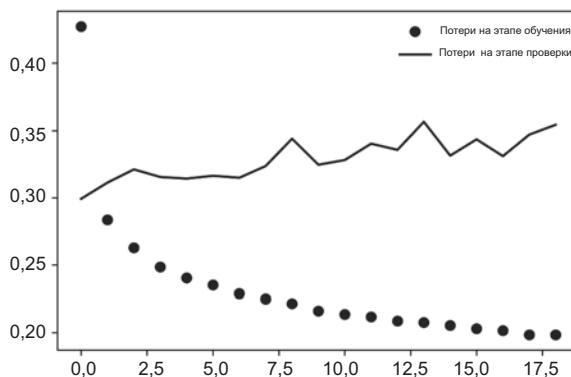
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(1, len(loss) + 1)

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



**Рис. 6.20.** Потери на этапах обучения и проверки простой полносвязной сети в задаче прогнозирования температуры по данным Jena

Некоторые значения потерь на этапе проверки близки к оценке ошибки базового решения без привлечения машинного обучения, но эта связь ненадежна. Это лишний раз показывает, как важно иметь базовое решение: в данном случае, как оказалось, его нелегко превзойти. Наше базовое решение основано на ценной информации, к которой нет доступа у модели машинного обучения.

Возможно, у вас появился вопрос: коль скоро существует хорошая и простая модель прогнозирования целей по имеющимся данным (базовое решение), почему обучаемая модель не смогла показать лучшие результаты? Потому что это простое решение совсем не то, что пытается найти обучаемая модель. Пространство моделей, в котором мы ищем решение, то есть пространство гипотез, — это пространство всех возможных двухслойных сетей с определяемой нами конфигурацией. Эти сети уже довольно сложны. Когда поиск решения выполняется в пространстве сложных моделей, простое базовое решение может оказаться ненадежным, даже если технически оно является частью пространства гипотез. Это существенное ограничение машинного обучения в целом: если алгоритм обучения не запрограммирован на поиск конкретной простой модели, обучение параметров иногда может терпеть неудачу в попытках найти простое решение простой задачи.

### 6.3.5. Первое базовое рекуррентное решение

Первое полносвязное решение не дало хорошего результата, но это не означает, что машинное обучение неприменимо к данной задаче. В предыдущем подходе первым действием было уменьшение размерности временных последовательностей, устранившее понятие времени из входных данных. Теперь посмотрим на эти данные как на то, чем они являются в действительности: последовательность, в которой важны причина и следствие. Попробуем использовать рекуррентную модель обработки последовательностей — она должна идеально подходить для таких последовательностей данных как раз потому, что, в отличие от первого решения, учитывает упорядочение образцов во времени.

Вместо слоя LSTM, представленного в предыдущем разделе, используем слой GRU, разработанный Чангом (Chung) с коллегами в 2014-м<sup>1</sup>. Слои управляемых рекуррентных блоков (Gated Recurrent Unit, GRU) основаны на том же принципе, что и слои LSTM, однако они представляют собой более простые структуры и, соответственно, менее затратны в вычислительном смысле (хотя могут не иметь такой же представительной мощности, как LSTM). Этот компромисс между затратностью вычислений и представительной мощностью можно наблюдать повсюду в области машинного обучения.

#### Листинг 6.39. Обучение и оценка модели на основе GRU

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

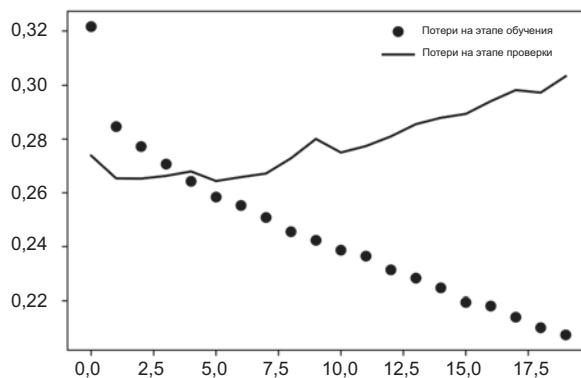
Результаты показаны на рис. 6.21. Они оказались намного лучше! Нам удалось значительно превзойти базовое решение и показать ценность машинного обучения, а также превосходство рекуррентных сетей над последовательными полносвязными сетями для решения подобных задач.

Новая оценка MAE ~0,265 (полученная до значительного проявления эффекта переобучения) после денормализации соответствует абсолютной ошибке 2,35 °C.

---

<sup>1</sup> Junyoung Chung et al., «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling», Conference on Neural Information Processing Systems (2014), <https://arxiv.org/abs/1412.3555>.

Это значительное улучшение относительно предыдущей ошибки  $2,57^{\circ}\text{C}$ , но у нас, возможно, еще есть потенциал для улучшения.



**Рис. 6.21.** Потери на этапах обучения и проверки модели на основе GRU в задаче прогнозирования температуры по данным Jena

### 6.3.6. Использование рекуррентного прореживания для борьбы с переобучением

Из кривых потерь на этапах обучения и проверки видно, что в модели быстро наступает эффект переобучения: потери начинают значительно отличаться уже после нескольких эпох. Вы уже знакомы с классическим приемом противостояния этому явлению — прореживанием, когда обнуляются случайно выбранные входные значения, чтобы разрушить неожиданные корреляции в обучающих данных, влияющих на слой. Однако правильное применение прореживания в рекуррентных сетях — сложная задача. Давно известно, что применение прореживания перед рекуррентным слоем скорее мешает обучению, а не помогает регуляризации. В 2015 году Ярин Гал (Yarin Gal), в рамках своей докторской диссертации по байесовскому глубокому обучению<sup>1</sup>, определил правильный способ применения прореживания к рекуррентным сетям: ко всем временным интервалам должна применяться одна и та же маска прореживания (должны обнуляться одни и те же значения) и не изменяться от интервала к интервалу. Более того, для регуляризации представлений, сформированных рекуррентными слоями, такими как GRU и LSTM, временно-постоянная маска прореживания должна применяться к внутренним рекуррентным активациям слоя (*рекуррентная маска прореживания*). Применение той же маски прореживания к каждому интервалу времени позволяет сети правильно распространить свою ошибку обучения во времени; временно-случайная маска нарушит этот сигнал ошибки и навредит процессу обучения.

<sup>1</sup> Yarin Gal, «Uncertainty in Deep Learning (PhD Thesis)», October 13, 2016, [http://mlg.eng.cam.ac.uk/yarin/blog\\_2248.html](http://mlg.eng.cam.ac.uk/yarin/blog_2248.html).

Ярин Гал провел исследования с использованием Keras и помог встроить этот механизм непосредственно в рекуррентные слои Keras. Каждый рекуррентный слой в Keras обладает двумя аргументами, имеющими отношение к прореживанию: `dropout`, вещественным числом, определяющим долю прореживаемых входных значений слоя, и `recurrent_dropout`, определяющим долю прореживаемых рекуррентных значений. Давайте добавим прореживание входных и рекуррентных значений в слой GRU и посмотрим, как это повлияет на переобучение. Поскольку сети, регуляризованные с применением прореживания, всегда требуют больше времени для полной сходимости, обучим сеть за количество эпох в два раза большее.

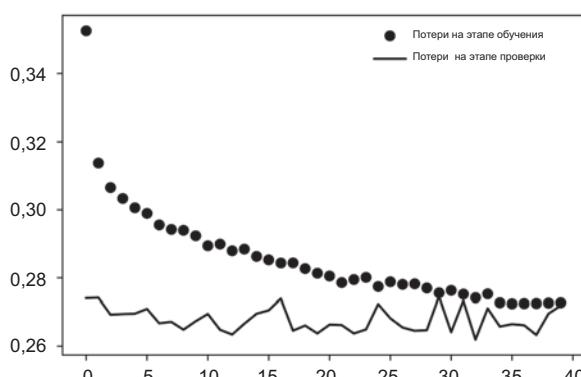
**Листинг 6.40.** Обучение и оценка модели на основе GRU с регуляризацией прореживанием

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.2,
                     recurrent_dropout=0.2,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=40,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Результаты показаны на рис. 6.22. Успех! Теперь эффект переобучения не наблюдается на протяжении первых 30 эпох. Однако хотя теперь оценки получились более стабильные, лучшие из них не намного ниже предыдущих.



**Рис. 6.22.** Потери на этапах обучения и проверки модели на основе GRU с прореживанием в задаче прогнозирования температуры по данным Jena

### 6.3.7. Наложение нескольких рекуррентных слоев друг на друга

Избавившись от эффекта переобучения, мы столкнулись с проблемой низкого качества, поэтому теперь нужно подумать об увеличении емкости сети. Вспомните описание обобщенного процесса машинного обучения: рекомендуется всегда стараться увеличивать емкость сети, пока на первое место не выйдет проблема переобучения (при условии, что предприняты все основные меры против нее, такие как прореживание). Пока проблема переобучения не стоит остро, вероятно, сеть имеет недостаточную емкость.

Увеличение емкости сети обычно осуществляется за счет увеличения числа параметров слоя или добавления дополнительных слоев. Наложение рекуррентных слоев друг на друга — классический способ конструирования более мощных рекуррентных сетей: например, в настоящее время алгоритм Google Translate представляет собой стек из семи больших слоев LSTM — это огромная сеть.

При наложении друг на друга рекуррентных слоев в Keras все промежуточные слои должны возвращать полные выходные последовательности (трехмерный тензор), а не только последний интервал. Это достигается установкой параметра `return_sequences=True`.

#### Листинг 6.41. Обучение и оценка модели с несколькими слоями GRU и с регуляризацией прореживанием

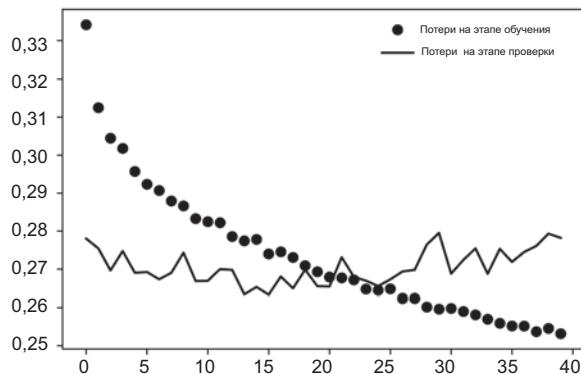
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=40,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Результаты показаны на рис. 6.23. Как видите, дополнительный слой немного улучшил результаты, хотя и незначительно. Отсюда можно сделать два вывода:

- Поскольку проблема переобучения все еще не стоит остро, можно продолжить увеличивать размеры слоев в попытках улучшить оценку потерь на этапе проверки. Однако это сопряжено с немалыми затратами вычислительных ресурсов.
- Добавление слоя не привело к существенному улучшению, то есть в данном случае наблюдается уменьшение отдачи от увеличения емкости сети.



**Рис. 6.23.** Потери на этапах обучения и проверки многослойной модели на основе GRU в задаче прогнозирования температуры по данным Jena

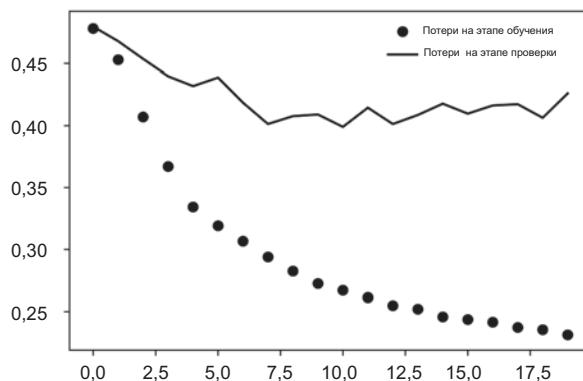
### 6.3.8. Использование двунаправленных рекуррентных нейронных сетей

Последнее средство, которое мы рассмотрим в этом разделе, называется *двунаправленные рекуррентные нейронные сети* (bidirectional RNN). Двунаправленная рекуррентная сеть – распространенная разновидность рекуррентных сетей, способная обеспечить более высокое качество решения некоторых задач. Она часто используется в обработке естественного языка – ее можно даже назвать швейцарским армейским ножом глубокого обучения для обработки естественного языка.

Рекуррентные сети зависят от порядка или от времени: они обрабатывают входные последовательности по порядку, и любое изменение порядка следования данных может полностью изменить представление, которое рекуррентная сеть извлечет из последовательности. Именно поэтому они так хорошо справляются с задачами, в которых порядок имеет значение, такими как задача прогнозирования температуры. Двунаправленная рекуррентная сеть использует чувствительность RNN к порядку: она состоит из двух обычных рекуррентных сетей, таких как слои GRU и LSTM, с которыми вы уже знакомы, каждая из этих сетей обрабатывает входную последовательность в одном направлении (прямом или обратном), и затем полученные представления объединяются. Обрабатывая последовательность в двух

направлениях, двунаправленная рекуррентная сеть способна выявить шаблоны, незаметные для односторонней сети.

Примечательно, что порядок обработки последовательностей в хронологическом порядке (от старых к новым) в этом разделе был выбран совершенно произвольно. По крайней мере, мы не пытались поставить это решение под вопрос. Могут ли рекуррентные сети показывать хорошие результаты, обрабатывая последовательности, например, в обратном порядке (от новых к старым)? Давайте попробуем применить это решение и посмотрим, что из этого получится. Для этого нужно лишь написать вариант генератора данных, обращающий входные последовательности (проще говоря, заменить последнюю строку в обобщенной функции-генераторе инструкцией `yield samples[:, ::-1, :], targets`). Обучение той же сети с единственным слоем GRU, которая использовалась в первом эксперименте в этом разделе, дало результаты, представленные на рис. 6.24.



**Рис. 6.24.** Потери на этапах обучения и проверки модели на основе GRU в задаче прогнозирования температуры по данным Йена с обучением на обращенных последовательностях

Сеть GRU, обрабатывающая последовательности в обратном порядке, не достигает даже уровня базового решения. Это явное свидетельство того, что в данном случае хронологический порядок обработки имеет большое значение для успеха. Это вполне объяснимо: слой GRU обычно запоминает недавнее прошлое лучше, чем более отдаленное, и, естественно, более свежая информация о погоде имеет большее значение для прогнозирования, чем старая (вот почему базовое решение без привлечения машинного обучения дает такую высокую точность). Поэтому версия слоя, обрабатывающая данные в прямом порядке, должна превосходить версию, обрабатывающую данные в обратном порядке. Важно отметить, что это не всегда верно для других задач, включая обработку естественных языков: очевидно, важность слова для понимания предложения обычно не зависит от его позиции в этом предложении. Давайте попробуем применить тот же прием к примеру анализа данных из IMDB со слоем LSTM из раздела 6.2.

**Листинг 6.42.** Обучение и оценка LSTM на обращенных последовательностях

Количество слов, рассматриваемых  
как признаки

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import layers
from keras.models import Sequential

→ max_features = 10000
maxlen = 500
```

Обрезка текста после этого  
количество слов (в числе  
max\_features самых распро-  
страненных слов)

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features) ← Загрузка данных
```

```
x_train = [x[::-1] for x in x_train] | Обращение последовательностей
x_test = [x[::-1] for x in x_test]
```

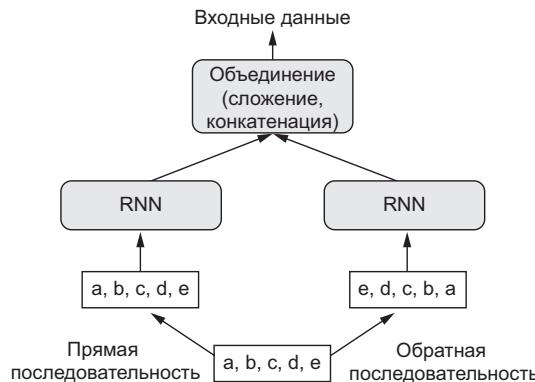
```
x_train = sequence.pad_sequences(x_train, maxlen=maxlen) | Дополнение после-
x_test = sequence.pad_sequences(x_test, maxlen=maxlen) | довательностей
```

```
model = Sequential()
model.add(layers.Embedding(max_features, 128))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)
```

В результате получилось качество, близкое к полученному с использованием слоя LSTM, обрабатывающего данные в хронологическом порядке. Примечательно, что с таким набором текстовых данных обработка последовательностей в обратном порядке дает такие же хорошие результаты, как обработка в прямом порядке. Это подтверждает гипотезу о том, что, хотя порядок следования слов в предложении важен для его понимания, направление обработки предложений *не имеет* решающего значения. Следует также отметить, что рекуррентная сеть, обученная на обращенных последовательностях, получит иные представления, так же как вы сами получили бы разные ментальные модели, если бы время текло в обратном направлении и вы проживали бы свою жизнь в направлении от смерти к рождению. В машинном обучении не следует пренебрегать *разными*, но *полезными* представлениями, и чем больше они различаются, тем лучше: они позволяют взглянуть на данные под другим углом, обнаружить аспекты, пропущенные другими подходами, и, как результат, улучшить качество решения задачи. Эта идея лежит в основе метода *обучения ансамблей*, который мы рассмотрим в главе 7.

Двунаправленная рекуррентная сеть использует эту идею для улучшения качества обучения на упорядоченных данных. Она просматривает входную последовательность в обоих направлениях (рис. 6.25), получает потенциально более насыщенные представления и выделяет шаблоны, которые могли быть упущены односторонней версией.



**Рис. 6.25.** Принцип действия двунаправленной рекуррентной нейронной сети

Для создания двунаправленной рекуррентной сети в Keras имеется слой `Bidirectional`, который в своем первом аргументе принимает экземпляр рекуррентного слоя. Слой `Bidirectional` создает второй, отдельный экземпляр этого рекуррентного слоя и использует один экземпляр для обработки входных последовательностей в прямом порядке, а другой — в обратном. Давайте опробуем этот прием на задаче анализа эмоциональной окраски отзывов в наборе данных IMDB.

#### Листинг 6.43. Обучение и оценка двунаправленной модели LSTM

```

model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train,
                     epochs=10,
                     batch_size=128,
                     validation_split=0.2)

```

Результаты немного улучшились в сравнении с обычной моделью LSTM, которую мы опробовали в предыдущем разделе. Оценка точности на этапе проверки превысила 89 %. Кроме того, похоже, что эта модель быстрее достигает состояния переобучения, что неудивительно, потому что двунаправленный слой имеет в два раза больше параметров по сравнению с односторонним слоем LSTM. При-

менив некоторую регуляризацию, мы наверняка смогли бы улучшить качество этой модели.

Теперь попробуем применить этот же подход к задаче прогнозирования температуры.

#### Листинг 6.44. Обучение двунаправленной модели GRU

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Bidirectional(
    layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=40,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Качество этой модели ничуть не улучшилось по сравнению с обычным слоем **GRU**. Легко понять почему: все прогностические способности исходят из половины сети, обрабатывающей данные в прямом хронологическом порядке, потому что, как мы уже выяснили, качество половины, обрабатывающей данные в обратном порядке, сильно отстает в этой задаче (потому что в данном случае недавнее прошлое имеет большее значение, чем отдаленное).

### 6.3.9. Что дальше

Существует множество других приемов, которые можно было бы попробовать применить, чтобы улучшить качество прогнозирования температуры:

- ❑ Изменить количество параметров в каждом рекуррентном слое в конфигурации с несколькими слоями. Текущий выбор был сделан практически произвольно и поэтому наверняка не является оптимальным.
- ❑ Изменить скорость обучения с помощью оптимизатора **RMSprop**.
- ❑ Попробовать использовать слои **LSTM** вместо **GRU**.
- ❑ Попробовать использовать больший полносвязный регрессор поверх рекуррентных слоев, то есть больший слой **Dense** или даже несколько слоев **Dense**.
- ❑ Не забудьте, наконец, опробовать лучшие модели (с точки зрения средней абсолютной ошибки) на контрольном наборе! Иначе у вас будут получаться архитектуры, переобученные на проверочном наборе.

Как всегда, глубокое обучение — это больше искусство, чем наука. Мы можем дать рекомендации, подсказав, какие приемы могут дать или не дать улучшение качества в данной задаче, но каждая задача в конечном счете уникальна; вам придется экспериментально оценить разные стратегии. В настоящее время нет теории, которая заранее сообщила бы, что следует сделать для получения оптимального решения задачи. Вы должны просто пробовать.

### 6.3.10. Подведение итогов

Вот какие выводы вы должны сделать из всего, что узнали в этом разделе:

- ❑ Как мы впервые узнали в главе 4, приступая к решению новой задачи, всегда желательно получить базовое решение, опираясь на метрики по вашему выбору. Если у вас не будет такого базового решения, на которое можно ориентироваться, вы не сможете сказать, движетесь ли вы в правильном направлении.
- ❑ Пробуйте сначала создавать простые модели, чтобы убедиться в необходимости приложения дополнительных усилий. Иногда простая модель может оказаться лучшим решением.
- ❑ Для обработки данных, в которых порядок следования имеет значение, лучше всего подходят рекуррентные сети — они с легкостью превосходят модели, которые сначала снижают размерность исходных данных.
- ❑ Применяя прием прореживания с рекуррентными сетями, используйте временно-постоянные и рекуррентные маски прореживания. В Keras уже имеются встроенные рекуррентные слои, поэтому вам останется только определить их аргументы `dropout` и `recurrent_dropout`.
- ❑ Комбинации из нескольких рекуррентных слоев обеспечивают большую представительность, чем один слой. Они также являются намного более затратными с точки зрения вычислений, и поэтому их применение не всегда оправдано. Они позволяют повысить качество решения сложных задач (таких, как машинный перевод), но не всегда подходят для небольших и простых задач.
- ❑ Двунаправленные рекуррентные сети, просматривающие последовательность данных в обоих направлениях, дают хорошие результаты в задачах обработки естественного языка. Но они мало пригодны для обработки последовательностей, в которых недавние данные информативнее, чем находящиеся в начале.

#### ПРИМЕЧАНИЕ

Существует еще два важных понятия, которые мы не будем подробно обсуждать здесь: рекуррентные модели с механизмом внимания (recurrent attention) и маскировка последовательностей (sequence masking). Оба способа хорошо подходят для обработки естественного языка и плохо — для прогнозирования температуры. Мы оставляем их вам для самостоятельного изучения.

## РЫНКИ И МАШИННОЕ ОБУЧЕНИЕ

Некоторые читатели наверняка захотят воспользоваться приемами, представленными здесь, для прогнозирования стоимости ценных бумаг на фондовом рынке (обменных курсов валют и т. д.). Рынки имеют *совершенно иные статистические характеристики*, чем природные явления, такие как погода. Использование машинного обучения для предсказания поведения рынков, когда имеются только общедоступные данные, — сложная задача, и вы, скорее всего, просто потратите силы и время, так ничего и не добившись.

Всегда помните: когда дело доходит до рынка, данные о прошлом служат плохой основой для предсказаний — невозможно двигаться вперед, глядя в зеркало заднего вида. С другой стороны, машинное обучение оправданно применять к наборам данных, когда прошлое служит хорошим предсказателем будущего.

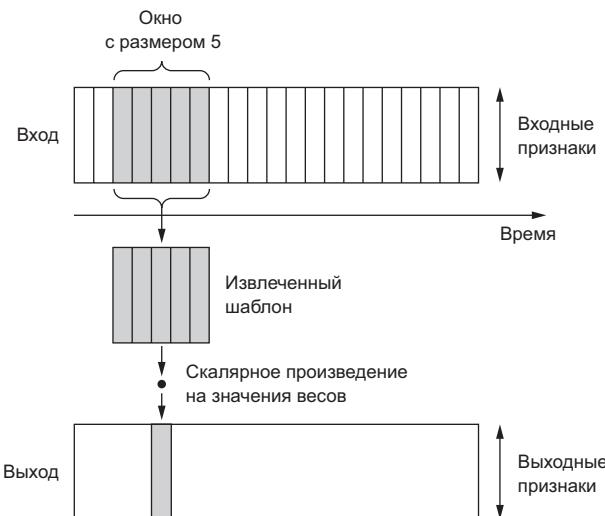
## 6.4. Обработка последовательностей с помощью сверточных нейронных сетей

В главе 5 мы познакомились со сверточными нейронными сетями и узнали, что они особенно хорошо подходят для решения задач распознавания образов благодаря их способности к *свертыванию* параметров, извлечению признаков из локальных входных шаблонов и получению эффективных и модульных представлений данных. Те же свойства сверточных нейронных сетей, которые делают их идеальным выбором для распознавания образов, прекрасно подходят для обработки последовательностей. Время можно рассматривать как пространственное измерение, подобно высоте или ширине двумерного изображения.

Такие одномерные сверточные нейронные сети с успехом могут состязаться с рекуррентными сетями в некоторых задачах обработки последовательностей, как правило, требуя меньше вычислительных ресурсов. Еще не так давно одномерные сверточные нейронные сети, которые обычно используются с расширенными ядрами, с успехом применялись для генерации звука и машинного перевода. В дополнение к этим конкретным достижениям давно известно, что небольшие одномерные сверточные нейронные сети могут служить быстрой альтернативой рекуррентным сетям в простых задачах, таких как классификация текста и прогнозирование временных последовательностей.

### 6.4.1. Обработка последовательных данных с помощью одномерной сверточной нейронной сети

Сверточные слои, с которыми мы познакомились выше, были двумерными свертками, извлекающими двумерные шаблоны из тензоров с изображениями и применяющими идентичные преобразования к каждому такому шаблону. Аналогично можно использовать одномерные свертки для извлечения одномерных шаблонов (подпоследовательностей) из последовательностей (рис. 6.26).



**Рис. 6.26.** Как действует одномерная сверточная нейронная сеть: каждый временной интервал извлекается из временного шаблона во входной последовательности

Такие одномерные сверточные слои способны распознавать локальные шаблоны в последовательности. Поскольку к каждому шаблону применяются одни и те же преобразования, тот или иной шаблон, найденный в некоторой позиции в предложении, позднее может быть опознан в другой позиции, что делает преобразования, выполняемые одномерными сверточными сетями, инвариантными (во времени). Например, одномерная сверточная сеть, обрабатывающая последовательность символов и использующая окно свертки с размером 5, способна запоминать слова или фрагменты слов длиной до 5 символов и распознавать эти слова в любом контексте во входной последовательности, — то есть одномерная сверточная сеть, обрабатывающая текст посимвольно, способна изучить морфологию слов.

## 6.4.2. Выбор соседних значений в одномерной последовательности данных

Вы уже знакомы с двумерными операциями выбора соседних значений, такими как выбор среднего или максимального значения из двумерных данных, которые используются в сверточных сетях для уменьшения разрешения тензоров с изображениями. Двумерные операции выбора имеют одномерный эквивалент, извлекающий одномерные шаблоны (подпоследовательности) из входных данных и возвращающий максимальное (выбор максимального значения из соседних) или среднее значение (выбор среднего значения из соседних). Так же как в двумерных сверточных сетях, эта операция используется для уменьшения длины одномерного входа (*снижение разрешения*).

### 6.4.3. Реализация одномерной сверточной сети

В Keras одномерные сверточные сети создаются с помощью слоя `Conv1D`, интерфейс которого напоминает интерфейс слоя `Conv2D`. Он принимает на входе трехмерные тензоры с формой (**образцы, время, признаки**) и возвращает трехмерные тензоры с той же формой. Окно свертки — это одномерное окно на оси времени: оси с индексом 1 во входном тензоре.

Попробуем сконструировать простую двухслойную одномерную сверточную сеть и использовать ее для решения уже знакомой нам задачи определения эмоциональной окраски отзывов в наборе данных IMDB. В качестве напоминания ниже приводится код получения и подготовки данных.

**Листинг 6.45.** Подготовка данных IMDB

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000
max_len = 500

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=
                                                       max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

Одномерные сверточные сети устроены так же, как их двумерные сородичи, с которыми мы познакомились в главе 5: они состоят из стопки слоев `Conv1D` и `MaxPooling1D`, завершающейся слоем `GlobalMaxPooling1D` или `Flatten`, который преобразует трехмерный вывод в двумерный, что позволяет добавить в модель один или несколько слоев `Dense` для классификации или регрессии.

Одно из различий — возможность использовать в одномерных сверточных сетях более крупные окна свертки. Окно свертки  $3 \times 3$  в двумерном сверточном слое содержит  $3 \times 3 = 9$  векторов признаков, а в одномерном сверточном слое окно с размером 3 содержит только 3 вектора признаков. Соответственно, мы легко можем использовать окна свертки с размером 7 или 9.

Вот пример одномерной сверточной нейронной сети для обработки набора данных IMDB.

**Листинг 6.46.** Обучение и оценка простой одномерной сверточной сети на данных IMDB

```

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

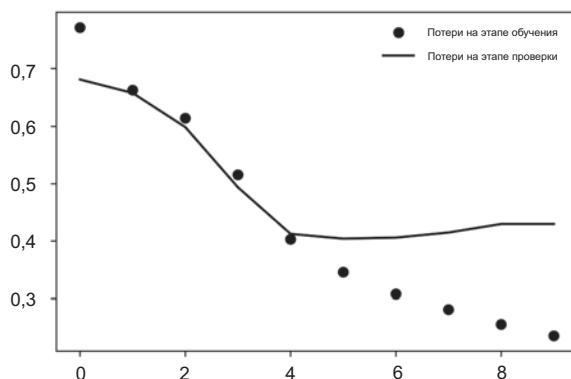
model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

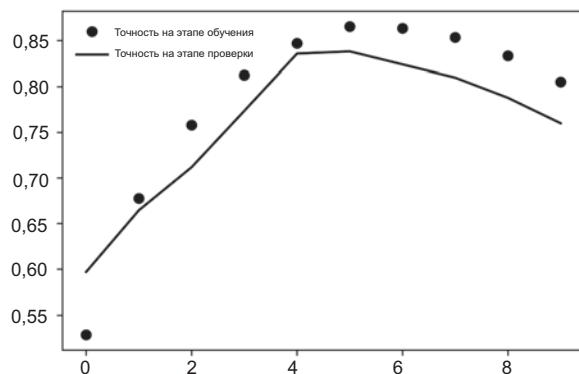
model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)

```

На рис. 6.27 и 6.28 показаны результаты обучения и проверки. Точность на этапе проверки получилась несколько ниже, чем с применением LSTM, но скорость обработки в обоих случаях — на CPU и GPU — оказалась выше (конкретное увеличение скорости в значительной степени зависит от аппаратной конфигурации). Теперь можно повторно обучить модель на правильно выбранном количестве эпох (четыре) и проверить ее на контрольном наборе данных. Это убедительная демонстрация того, что одномерная сверточная нейронная сеть может служить быстрой и недорогой альтернативой рекуррентной сети в задаче определения эмоциональной окраски текста.



**Рис. 6.27.** Потери на этапах обучения и проверки для модели анализа IMDB с простой одномерной сверточной сетью



**Рис. 6.28.** Точность на этапах обучения и проверки для модели анализа IMDB с простой одномерной сверточной сетью

#### 6.4.4. Объединение сверточных и рекуррентных сетей для обработки длинных последовательностей

Поскольку одномерная сверточная сеть обрабатывает входные шаблоны независимо, она нечувствительна к порядку следования временных интервалов (кроме как в локальном масштабе, в пределах окна свертки), в отличие от рекуррентной сети. Конечно, для распознавания более протяженных шаблонов можно использовать прием наложения друг на друга нескольких сверточных слоев и слоев выбора соседних значений, в результате верхние слои будут видеть все более длинные фрагменты исходных входных данных, однако это слишком слабый способ включить чувствительность к порядку. Доказать слабость этого подхода можно на примере применения одномерных сверточных сетей для решения задачи прогнозирования температуры, где чувствительность к порядку является ключевым фактором для получения хорошего прогноза. В следующем примере повторно используются переменные, которые были определены прежде: `float_data`, `train_gen`, `val_gen` и `val_steps`.

**Листинг 6.47.** Обучение и оценка простой одномерной сверточной сети на данных из набора Jena

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                      input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
```

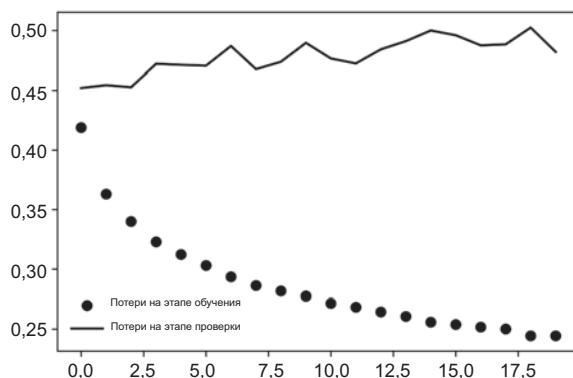
```

model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)

```

На рис. 6.29 показаны графики потерь по средней абсолютной ошибке (MAE) на этапах обучения и оценки.

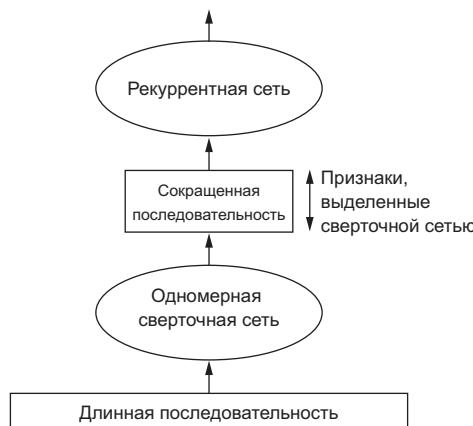


**Рис. 6.29.** Потери на этапах обучения и проверки модели на основе простой одномерной сверточной сети в задаче прогнозирования температуры по данным Jena

Величина средней абсолютной ошибки на этапе проверки остается выше 0,40: маленькой сверточной сети не удалось превзойти даже базовое решение. И снова причина в том, что сверточная сеть отыскивает шаблоны в любом месте во входной последовательности без учета их местоположения (ближе к началу, ближе к концу и т. д.). Так как в данной задаче прогнозирования недавние данные должны интерпретироваться иначе, чем отдаленные, сверточной сети не удалось произвести значимые результаты. Однако это ограничение сверточных сетей не является проблемой для задачи классификации отзывов в наборе данных IMDB, потому что ключевые шаблоны, ассоциирующиеся с положительной или отрицательной эмоциональной окраской, остаются информативными, независимо от их местоположения во входных последовательностях.

Одна из стратегий объединить скорость и легкость сверточных сетей с чувствительностью к порядку рекуррентных сетей заключается в использовании одномерной сверточной сети для предварительной обработки данных перед передачей их в рекуррентную сеть (рис. 6.30). Этот прием оказывается особенно выгодным, когда имеющиеся последовательности настолько длинны (с несколькими тысячами

ми интервалов и больше), что их нереально обработать с помощью рекуррентной сети. Сверточная часть превратит длинную входную последовательность в более короткую последовательность высокого уровня признаков (уменьшив ее разрешение). А затем последовательность выделенных признаков подается на вход рекуррентной части сети.



**Рис. 6.30.** Объединение одномерной сверточной и рекуррентной сетей для обработки длинных последовательностей

Этот прием нечасто можно встретить в научных статьях и практических приложениях, возможно, потому, что он мало известен. Однако он имеет довольно высокую эффективность и заслуживает более широкого распространения. Опробуем его на задаче прогнозирования температуры. Так как эта стратегия позволяет манипулировать гораздо более длинными последовательностями, мы можем задействовать более отдаленные данные (увеличив параметр `lookback` генератора данных) или увеличить разрешение временных последовательностей (уменьшив параметр `step` генератора). Для данного примера произвольно было решено уменьшить значение `step` наполовину, в результате чего временные последовательности увеличились вдвое, и теперь образцы со значениями температуры отбираются с 30-минутным интервалом. В примере повторно используется функция-генератор, которая была определена выше (см. листинг 6.33).

#### Листинг 6.48. Подготовка генераторов данных с высоким разрешением для набора данных Jena

```

step = 3
lookback = 720
delay = 144

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
  
```

← Премя имело значение 6 (один образец для каждого часа);  
теперь имеет значение 3 (один образец для каждого 30 минут)

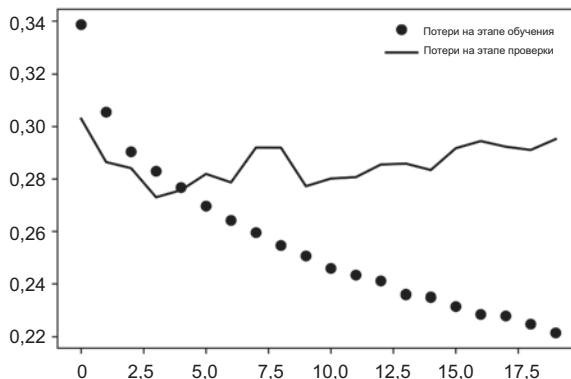
← Не изменились

```

        min_index=0,
        max_index=200000,
        shuffle=True,
        step=step)
val_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=200001,
                     max_index=300000,
                     step=step)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step)
val_steps = (300000 - 200001 - lookback) // 128
test_steps = (len(float_data) - 300001 - lookback) // 128

```

Модель начинается с двух уровней Conv1D, за которыми следует уровень GRU. Результаты работы модели показаны на рис. 6.31.



**Рис. 6.31.** Потери на этапах обучения и проверки модели на основе одномерной сверточной сети и слоя GRU в задаче прогнозирования температуры по данным Jena

**Листинг 6.49.** Модель, объединяющая одномерную сверточную основу и уровень GRU

```

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                      input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))

```

Продолжение ↗

**Листинг 6.49** (продолжение)

```
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)
```

Судя по величине потерь на этапе проверки, эта комбинация не дотягивает до решения с регуляризованным слоем GRU, зато она действует намного быстрее. Она просматривает вдвое больше данных, что в этом случае не кажется особенно полезным, но может быть важным в других задачах.

#### 6.4.5. Подведение итогов

Вот какие выводы вы должны сделать из всего, что узнали в этом разделе:

- ❑ По аналогии с двумерными сверточными сетями, которые прекрасно справляются с задачей выделения визуальных шаблонов в двумерном пространстве, одномерные сверточные сети хорошо подходят для выделения временных шаблонов. В некоторых задачах, особенно в обработке естественного языка, они могут служить более быстрой альтернативой рекуррентным сетям.
- ❑ Обычно одномерные сверточные сети структурируются так же, как их двумерные сородичи из мира распознавания образов: они состоят из стопки слоев Conv1D и MaxPooling1D, завершающейся слоем GlobalMaxPooling1D или Flatten.
- ❑ Поскольку применение рекуррентных сетей является слишком затратным для обработки очень длинных последовательностей, а применение одномерных сверточных сетей — менее затратным, может оказаться неплохой идея использовать одномерную сверточную сеть для предварительной обработки последовательности перед передачей в рекуррентную сеть. Она сократит последовательность и выделит полезное представление для последующей обработки рекуррентной сетью.

### Краткие итоги главы

- ❑ В этой главе вы познакомились со следующими приемами, применимыми к любым наборам данных, от текста до временных последовательностей:
  - как токенизировать текст;
  - что такое векторные представления слов и как их использовать;

- что такое рекуррентные сети и как их использовать;
  - как составлять комбинации рекуррентных слоев и использовать двунаправленные рекуррентные сети для создания мощных моделей обработки последовательностей;
  - как использовать одномерные сверточные сети для обработки последовательностей;
  - как объединять одномерные сверточные и рекуррентные сети для обработки длинных последовательностей.
- Рекуррентные сети можно использовать для регрессии («прогнозирования будущего»), классификации, выделения аномалий и маркировки последовательностей (например, для выделения имен или дат в предложениях).
- Аналогично одномерные сверточные сети можно использовать для реализации машинного перевода (сверточные модели преобразования последовательностей в последовательности (*sequence-to-sequence*), такие как SliceNeta<sup>1</sup>), классификации документов и проверки орфографии.
- Если *глобальный порядок следования данных в последовательности имеет значение*, обрабатывать такие данные предпочтительнее с применением рекуррентной сети. Это относится, например, к временным последовательностям, в которых недавнее прошлое более информативно, чем отдаленное.
- Если *глобальный порядок следования данных не имеет решающего значения*, для их обработки с успехом можно использовать одномерные сверточные сети, применение которых менее затратно, а полученный результат оказывается по крайней мере не хуже. Это в особенности относится к текстовым данным, где ключевой шаблон, найденный в начале предложения, будет не менее значимым, чем шаблон, найденный в конце.

<sup>1</sup> См. <https://arxiv.org/abs/1706.03059>.

# 7

# Лучшие практики глубокого обучения продвинутого уровня

Эта глава охватывает следующие темы:

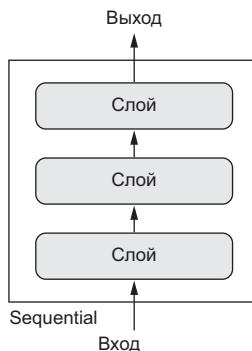
- ✓ функциональный API фреймворка Keras;
- ✓ использование обратных вызовов Keras;
- ✓ использование инструмента визуализации TensorBoard;
- ✓ важнейшие лучшие практики для разработки современных моделей.

В этой главе рассматривается несколько мощных инструментов, позволяющих конструировать самые современные модели для решения сложных задач. Используя функциональный API фреймворка Keras, вы сможете строить графоподобные модели, повторно задействовать один и тот же слой для обработки разных входов и использовать модели Keras подобно функциям на языке Python. Обратные вызовы Keras и инструмент визуализации TensorBoard позволяют следить за процессом обучения моделей. Также мы обсудим некоторые другие продвинутые приемы, включая пакетную нормализацию, остаточные связи, оптимизацию гиперпараметров и ансамблирование.

## 7.1. За рамками модели Sequential: функциональный API фреймворка Keras

Все нейронные сети, представленные выше в этой книге, были реализованы с применением модели `Sequential`. Эта модель основана на предположении, что сеть имеет только один вход и только один выход и состоит из линейного стека слоев (рис. 7.1).

Это общепринятое предположение; данная конфигурация настолько распространена, что до настоящего момента мы смогли охватить множество тем и практических

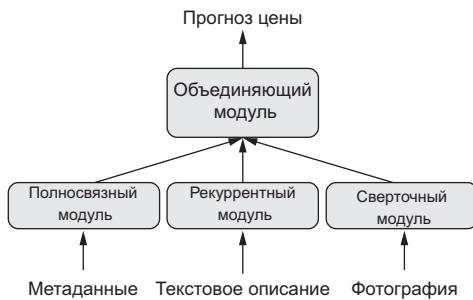


**Рис. 7.1.** Модель Sequential: линейный стек слоев

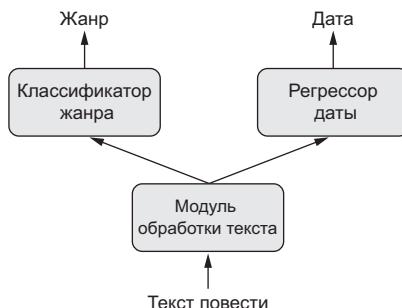
применений, использовав только класс `Sequential` моделей. Однако в ряде случаев это предположение оказывается слишком жестким. Некоторые сети требуют нескольких независимых входов, другие — нескольких выходов, а третий имеют внутренние ветви, соединяющие слои, что делает их похожими на *графы*, а не на линейные стеки слоев.

Некоторые задачи, например, требуют мультимодальных входов: они объединяют данные, поступающие из разных источников, обрабатывая каждый вид данных с использованием разных типов нейронных слоев. Представьте модель глубокого обучения, которая пытается предсказать наиболее вероятную рыночную цену подержанной одежды, используя следующие входные данные: метаданные, представленные пользователем (такие, как торговая марка производителя, срок использования и т. д.), текстовое описание и фотографию. Если бы у вас имелись только метаданные, вы могли бы применить к ним метод прямого кодирования и использовать для предсказания цены полносвязную сеть. Если бы у вас имелось только текстовое описание, вы могли бы использовать рекуррентную или одномерную сверточную сеть. Если бы у вас имелась только фотография, вы могли бы использовать двумерную сверточную сеть. Но как использовать все три вида входных данных одновременно? Простейшим решением могло бы быть обучение трех отдельных моделей с последующим вычислением взвешенного среднего их предсказаний. Однако такое решение может оказаться не самым оптимальным, потому что информация, извлекаемая моделями, может быть избыточной. Лучшим решением является изучение более точной модели данных с использованием модели машинного обучения, которая способна обрабатывать все входные модальности одновременно, — модели с тремя входными ветвями (рис. 7.2).

Аналогично, в некоторых задачах требуется предсказать несколько целевых атрибутов по входным данным. Например, по тексту повести или рассказа определить жанр (любовный роман или военная повесть) и примерную дату их написания. Конечно, можно подготовить две отдельные модели: одну для определения жанра, а другую — для даты. Однако поскольку эти атрибуты не являются статистически независимыми, лучшим решением будет создать модель, обучающуюся для одновременного

**Рис. 7.2.** Модель с несколькими входами

предсказания обоих выходных атрибутов — жанра и даты. Такая объединенная модель будет иметь два выхода, или *головы* (рис. 7.3). Благодаря корреляции между жанром и датой, знание даты может помочь модели получить богатое и точное представление пространства жанров литературных произведений, и наоборот.

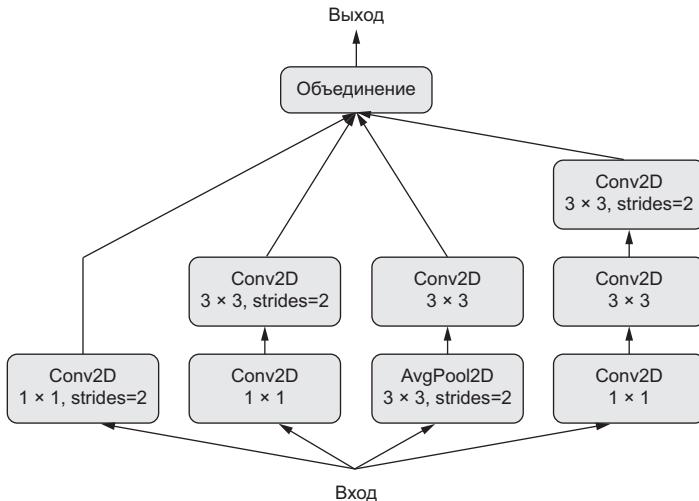
**Рис. 7.3.** Модель с несколькими выходами (головами)

Кроме того, многие нейронные архитектуры, разработанные недавно, требуют нелинейной организации сети, когда сеть имеет структуру ориентированного ациклического графа. Семейство сетей Inception (разработанное Кристианом Сегеди (Christian Szegedy) с коллегами в Google)<sup>1</sup>, например, опирается на *модули Inception*, в которых входные данные обрабатываются несколькими параллельными сверточными ветвями, выходы которых затем объединяются в единый тензор (рис. 7.4). Также недавно появилась методика добавления в модель *остаточных связей*, развитие которой началось с появления семейства сетей ResNet (разработанного Каймином Хе (Kaiming He) с коллегами в Microsoft)<sup>2</sup>. Суть этого приема заключается в повторном внедрении предыдущих представлений в исходящий поток данных добавлением про-

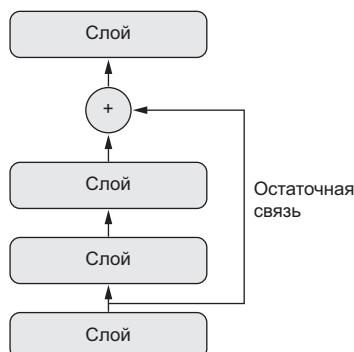
<sup>1</sup> Christian Szegedy et al., «Going Deeper with Convolutions», Conference on Computer Vision and Pattern Recognition (2014), <https://arxiv.org/abs/1409.4842>.

<sup>2</sup> Kaiming He et al., «Deep Residual Learning for Image Recognition», Conference on Computer Vision and Pattern Recognition (2015), <https://arxiv.org/abs/1512.03385>.

шлого выходного тензора к более новому выходному тензору (рис. 7.5), что помогает предотвратить потерю информации в процессе обработки данных. Существует также множество других примеров, таких как графоподобные сети.



**Рис. 7.4.** Модуль Inception: подграф уровней с несколькими параллельными сверточными ветвями



**Рис. 7.5.** Остаточные связи: повторное внедрение предыдущей исходящей информации добавлением в карту признаков

Эти три важные разновидности моделей — модели с несколькими входами, модели с несколькими выходами и графоподобные модели — невозможно реализовать с использованием только класса `Sequential` моделей из фреймворка Keras. Однако существует другой, намного более универсальный и гибкий способ использования Keras — *функциональный API*. В этом разделе подробно рассказывается, что это такое, описываются его возможности и особенности использования.

### 7.1.1. Введение в функциональный API

Функциональный API позволяет напрямую манипулировать тензорами и использовать уровни как функции, которые принимают и возвращают тензоры (чем и обусловлено такое название — *функциональный API*):

```
from keras import Input, layers
input_tensor = Input(shape=(32,)) ← Тензор
dense = layers.Dense(32, activation='relu') ← Слой — это функция
output_tensor = dense(input_tensor) ←
    Вызываемый слой может принимать
    и возвращать тензор
```

Начнем с маленького примера, демонстрирующего простую модель `Sequential` и ее эквивалент с использованием функционального API:

```
from keras.models import Sequential, Model
from keras import layers
from keras import Input

seq_model = Sequential() ← Уже знакомая нам модель Sequential
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))

input_tensor = Input(shape=(64,)) ← Ее функциональный
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x) ← эквивалент

model = Model(input_tensor, output_tensor) ← Класс Model превращает входной
model.summary() ← и выходной тензоры в модель
    Рассмотрим ее!
```

Вот что вывел вызов `model.summary()`:

---

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64)	0
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 32)	1056
dense_3 (Dense)	(None, 10)	330

---

Total params: 3,466  
Trainable params: 3,466  
Non-trainable params: 0

---

Единственное, что может показаться здесь немного необычным, — это создание экземпляра класса `Model` только с использованием входного и выходного тензоров. За кулисами Keras извлекает все слои, участвовавшие в преобразовании тензора `input_tensor` в тензор `output_tensor`, и объединяет их в графоподобную структуру данных — `Model`. Конечно, подобное возможно только для выходного тензора `output_tensor`, полученного путем многоократных преобразований входного тензора `input_tensor`. Если попытаться создать модель из не связанных между собой входов и выходов, вы получите исключение `RuntimeError`:

```
>>> unrelated_input = Input(shape=(32,))
>>> bad_model = Model(unrelated_input, output_tensor)
RuntimeError: Graph disconnected: cannot
obtain value for tensor
↳Tensor("input_1:0", shape=(?, 64), dtype=float32) at layer "input_1".
```

Это исключение фактически сообщает, что фреймворку Keras не удалось достичь `input_1` из переданного ему выходного тензора.

Компиляция, обучение и оценка такого экземпляра `Model` выглядит точно так же, как при использовании модели `Sequential`:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy') ← Компиляция модели
import numpy as np ← Генерация фиктивных данных для обучения
x_train = np.random.random((1000, 64))
y_train = np.random.random((1000, 10))

model.fit(x_train, y_train, epochs=10, batch_size=128) ← Обучение модели на протяжении 10 эпох

score = model.evaluate(x_train, y_train) ← Оценка модели
```

## 7.1.2. Модели с несколькими входами

Функциональный API можно использовать для создания моделей с несколькими входами. Обычно такие модели в какой-то момент объединяют свои входные ветви, используя слой, способный объединить несколько тензоров: сложением, слиянием или как-то иначе. Часто для этого используются операции слияния, реализованные в Keras, такие как `keras.layers.add`, `keras.layers.concatenate` и т. д. Рассмотрим пример очень простой модели с несколькими входами: модель вида «вопрос/ответ».

Типичная модель «вопрос/ответ» имеет два входа: вопрос на естественном языке и фрагмент текста (например, новостная статья) с информацией, которая будет использоваться для ответа на вопрос. Опираясь на эти данные, модель должна вернуть ответ: в простейшем случае это может быть ответ, состоящий из одного слова, полученного применением классификатора `softmax` к некоторому предопределенному словарю (рис. 7.6).

Следующий пример демонстрирует, как создать модель с помощью функционального API. В нем создаются две независимые ветви, входной текст и вопрос коди-

руются в векторные представления; затем эти векторы объединяются и, наконец, поверх объединенного представления добавляется классификатор softmax.



**Рис. 7.6.** Модель «вопрос/ответ»

**Листинг 7.1.** Реализация модели «вопрос/ответ» с двумя входами с использованием функционального API

```

from keras.models import Model
from keras import layers
from keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text') ←
  Входной текст — это последова-
  тельность целых чисел переменной
  длины. Обратите внимание на то,
  что при желании можно задать имя
  последовательности

embedded_text = layers.Embedding(
    text_vocabulary_size, 64)(text_input) ←
  Преобразование входного текста в после-
  довательность векторов с размером 64

encoded_text = layers.LSTM(32)(embedded_text) ←
  Преобразование векторов в единый
  вектор с помощью уровня LSTM

question_input = Input(shape=(None,), ←
  dtype='int32',
  name='question') ←
  Та же процедура (с другими экземпляра-
  ми слоев) повторяется для вопроса

embedded_question = layers.Embedding(
    question_vocabulary_size, 32)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
  axis=-1) ←
  Объединение закодированных
  вопроса и текста
  
```

```

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated) ←

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc']) ←

```

Создание экземпляра модели с двумя входами и одним выходом

Добавление классификатора softmax сверху

Теперь встает вопрос обучения модели с двумя входами. Для этого можно использовать два разных API: можно передать модели список массивов NumPy или словарь, отображающий имена входов в массивы NumPy. Естественно, последний вариант возможен, только если вы определили имена для входов.

### Листинг 7.2. Передача данных в модель с несколькими входами

```

import numpy as np

num_samples = 1000 ← Создание массива NumPy
max_length = 100 ← с фиктивными данными

text = np.random.randint(1, text_vocabulary_size, ←
                        size=(num_samples, max_length))
question = np.random.randint(1, question_vocabulary_size, ←
                            size=(num_samples, max_length)) ←

К вопросам применяется прямое кодирование, а не преобразование в целые числа

```

```

answers = np.zeros(shape=(num_samples, answer_vocabulary_size))
indices = np.random.randint(0, answer_vocabulary_size, size=num_samples)
for i, x in enumerate(answers):
    x[indices[i]] = 1

model.fit([text, question], answers, epochs=10, batch_size=128) ←

model.fit({'text': text, 'question': question}, answers, ←
          epochs=10, batch_size=128) ←

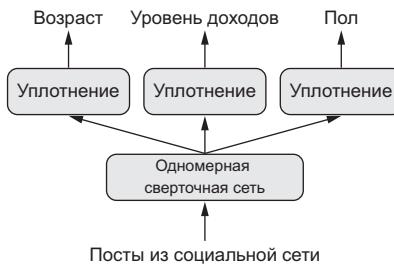
```

Передача списка входов

Передача с помощью словаря (возможна, только если были определены имена для входов)

### 7.1.3. Модели с несколькими выходами

Функциональный API также можно использовать для создания моделей с несколькими выходами (или *головами*). Простейшим примером может служить сеть, пытающаяся одновременно предсказать разные свойства данных, например принимающая на входе последовательность постов из социальной сети от некоторой анонимной персоны и пытающаяся предсказать характеристики этой персоны, такие как возраст, пол и уровень доходов (рис. 7.7).

**Рис. 7.7.** Модель «вопрос/ответ»**Листинг 7.3.** Реализация модели с тремя выходами с использованием функционального API

```

from keras import layers
from keras import Input
from keras.models import Model

vocabulary_size = 50000
num_income_groups = 10

posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)

age_prediction = layers.Dense(1, name='age')(x) ← Обратите внимание: для выход-
income_prediction = layers.Dense(num_income_groups,     ых слов определены имена
                                 activation='softmax',
                                 name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(posts_input,
              [age_prediction, income_prediction, gender_prediction])
  
```

Важно отметить, что для обучения такой модели необходима возможность задавать разные функции потерь для разных выходов: например, определение возраста — это задача скалярной регрессии, но определение пола — задача бинарной классификации, требующая отдельной процедуры обучения. Однако из-за того, что градиентный спуск требует минимизации скаляра, эти функции потерь должны объединяться в единственное значение. Простейший способ объединения потерь — их суммирование. В Keras для этого можно передать функции `compile` список или словарь с разными объектами для разных выходов; в результате значения потерь будут суммироваться в общее значение потери, которое будет минимизироваться в ходе обучения.

**Листинг 7.4.** Параметры компиляции модели с несколькими выходами:  
несколько функций потерь

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'])

model.compile(optimizer='rmsprop',
              loss={'age': 'mse',
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'})
```

Эквивалентное решение  
(возможно, только если  
определены имена  
выходных слоев)

Обратите внимание: несбалансированные вклады потерь приведут к созданию представления, оптимизированного преимущественно для задачи с наибольшей потерей, в ущерб другим задачам. Чтобы исправить эту проблему, можно присвоить разные степени важности значениям потерь, вносящим вклад в общую потерю. Это может пригодиться, когда значения потерь имеют разные масштабы. Например, средняя квадратичная ошибка (Mean Squared Error, MSE), используемая как функция потерь в задаче определения возраста, обычно принимает значение около 3–5, тогда как перекрестная энтропия, используемая в задаче определения пола, может колебаться около величины 0,1. Чтобы в такой ситуации сбалансировать вклад разных потерь, можно присвоить вес 10 перекрестной энтропии и вес 0,25 — оценке MSE.

**Листинг 7.5.** Параметры компиляции модели с несколькими выходами:  
взвешивание потерь

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'],
              loss_weights=[0.25, 1., 10.])

model.compile(optimizer='rmsprop',
              loss={'age': 'mse',
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'},
              loss_weights={'age': 0.25,
                           'income': 1.,
                           'gender': 10.})
```

Эквивалентное решение  
(возможно, только если  
определены имена  
выходных слоев)

Так же как в случае с моделями, имеющими несколько входов, передавать обучающие данные в модель можно либо в виде списка массивов NumPy, либо в виде словаря с их именами.

**Листинг 7.6.** Передача данных в модель с несколькими выходами

```
model.fit(posts, [age_targets, income_targets, gender_targets],
          epochs=10, batch_size=64)

model.fit(posts, {'age': age_targets,
                  'income': income_targets,
                  'gender': gender_targets},
          epochs=10, batch_size=64)
```

Эквивалентное решение  
(возможно, только если  
определены имена  
выходных слоев)

Предполагается, что `age_targets`, `income_targets` и `gender_targets` — это массивы NumPy

## 7.1.4. Ориентированные ациклические графы уровней

С помощью функционального API можно не только создавать модели с несколькими входами и выходами, но также конструировать сети со сложной внутренней топологией. Фреймворк Keras позволяет создавать нейронные сети, организованные как произвольные *ориентированные ациклические графы* слоев. Уточнение *ациклические* очень важно: такие графы не имеют замкнутых маршрутов. Тензор  $x$  не может служить входом в один из слоев, генерирующих  $x$ . Единственными допустимыми циклами обработки (то есть рекуррентными связями) являются циклы внутри рекуррентных слоев.

Существует несколько типичных компонентов нейронных сетей, реализуемых как графы. Наиболее известными из них являются модули Inception и остаточные связи. Давайте посмотрим, как оба они реализованы в Keras, чтобы лучше понять, как с помощью функционального API можно создавать графы слоев.

### Модули Inception

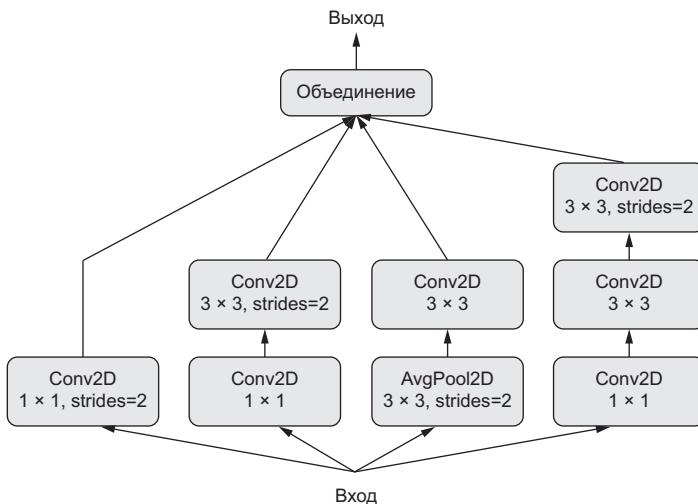
Под названием *Inception*<sup>1</sup> подразумевается популярная архитектура сверточных нейронных сетей; она была разработана Кристианом Сегеди (Christian Szegedy) и его коллегами в компании Google в 2013–2014 годах, вдохновленными архитектурой *сеть в сети*, разработанной ранее<sup>2</sup>. Она состоит из стека модулей, которые сами имеют вид небольших и независимых сетей, разбитого на несколько параллельных ветвей. В самом простом виде модуль Inception имеет три-четыре ветви, начинающиеся сверткой  $1 \times 1$ , за которой следует свертка  $3 \times 3$ , и заканчивающиеся объединением выделенных признаков. Такая организация помогает сети выделять пространственные и канальные признаки отдельно, что обеспечивает более высокую эффективность, чем при совместном их извлечении. Возможны также более сложные версии модуля Inception, которые обычно включают в себя операции объединения, разные размеры пространственной свертки (например,  $5 \times 5$  вместо  $3 \times 3$  в некоторых ветвях) и ветви без пространственной свертки (то есть включающие в себя только свертку  $1 \times 1$ ). Пример такого модуля, взятого из Inception V3, показан на рис. 7.8.

### НАЗНАЧЕНИЕ СВЕРТОК $1 \times 1$

Вы уже знаете, что свертки извлекают пространственные шаблоны вокруг каждой клетки во входном тензоре и применяют к ним одни и те же преобразования. Пограничный случай — когда извлекаемые шаблоны состоят из одной клетки. В этом случае операция свертки эквивалентна обработке вектора каждой клетки слоем Dense: она вычисляет признаки, смешивающие информацию из каналов во входном тензоре, но не смешивает информацию, распределенную в пространстве (потому что каждый раз обрабатывает только одну клетку). Такие свертки  $1 \times 1$  (также называемые *поточечными свертками*) используются в модулях Inception, где помогают отделить друг от друга канальные

<sup>1</sup> <https://arxiv.org/abs/1409.4842>.

<sup>2</sup> Min Lin, Qiang Chen, and Shuicheng Yan, «Network in Network», International Conference on Learning Representations (2013), <https://arxiv.org/abs/1312.4400>.



**Рис. 7.8.** Модуль Inception

и пространственные признаки, что очень разумно, если предполагается, что каждый канал имеет сильную автокорреляцию в пространстве, но разные каналы могут слабо коррелировать друг с другом.

Вот как можно реализовать модуль, изображенный на рис. 7.8, с помощью функционального API. Этот пример предполагает наличие четырехмерного входного тензора  $x$ :

Все ветви имеют одинаковый шаг свертки (2). Это необходимо для получения на выходе всех ветвей тензоров одного размера, чтобы потом их можно было объединить

```
from keras import layers

branch_a = layers.Conv2D(128, 1,
>                      activation='relu', strides=2)(x)
branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate(
    [branch_a, branch_b, branch_c, branch_d], axis=-1)
```

В этой ветви шагание происходит в слое извлечения среднего по соседям

## Объединение результатов, возвращаемых ветвями для получения результата работы модуля

Обратите внимание на то, что полная архитектура Inception V3 доступна в Keras как `keras.applications.inception_v3.InceptionV3`, включая веса, полученные предварительным обучением на наборе данных ImageNet. В модуле `applications`, во фреймворке Keras, имеется еще одна похожая модель — *Xception*<sup>1</sup>. Название *Xception* происходит от *extreme inception*. Это архитектура сверточных сетей, разработанная отчасти под влиянием Inception. Ее идея состоит в том, чтобы полностью разделить выделение канальных и пространственных признаков и заменить модули Inception раздельными свертками по глубине (*depthwise separable convolution*), состоящими из глубоких сверток (пространственных сверток, в которых каждый входной канал обрабатывается отдельно), за которыми следуют поточечные свертки (свертки  $1 \times 1$ ) — фактически крайняя форма модуля Inception, в которой пространственные и канальные признаки полностью разделены. Xception имеет примерно такое же число параметров, как Inception V3, но показывает более высокую скорость работы и точность на наборе ImageNet, а также на других больших наборах данных благодаря более эффективному использованию параметров модели.

## Остаточные связи

*Остаточные связи* — это распространенный графоподобный компонент сети, который можно встретить во многих архитектурах сетей, разработанных после 2015 года, включая Xception. Они были предложены Каймином Хе (Kaiming He) с коллегами из Microsoft в их победной работе, выигравшей состязание ILSVRC ImageNet в конце 2015 года<sup>2</sup>. Остаточные связи решают две распространенные проблемы, затрагивающие любые крупномасштабные модели глубокого обучения: затухание градиентов и недостаточную репрезентативность. В общем случае добавление остаточных связей в любую модель, имеющую более 10 слоев, почти наверняка даст положительный результат.

Остаточная связь заключается в передаче вывода более раннего слоя на вход более позднего слоя, вследствие чего в последовательной сети фактически создается короткий путь. Вместо объединения с более поздней активацией вывод, полученный ранее, суммируется с более поздней активацией, что предполагает равенство размеров обеих активаций. Если они имеют разные размеры, можно применить линейное преобразование для приведения формы ранней активации к форме цели (например, слой `Dense` без активации или, для карт сверточных признаков, свертку  $1 \times 1$  без активации).

Вот как можно реализовать остаточную связь в Keras, когда размеры карт признаков совпадают. Этот пример предполагает наличие четырехмерного входного тензора `x`:

---

<sup>1</sup> François Chollet, «Xception: Deep Learning with Depthwise Separable Convolutions», Conference on Computer Vision and Pattern Recognition (2017), <https://arxiv.org/abs/1610.02357>.

<sup>2</sup> He et al., «Deep Residual Learning for Image Recognition», <https://arxiv.org/abs/>.

```
from keras import layers
x = ...
y = layers.Conv2D(128, 3, activation='relu', padding='same')(x) ← Применение
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y) ← преобразования к x
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)

y = layers.add([y, x]) ← Добавление оригинального
                        тензора x к выходным признакам
```

А вот так реализуется остаточная связь, когда размеры карт признаков различаются (и снова предполагается наличие четырехмерного входного тензора x):

```
from keras import layers
x = ...
y = layers.Conv2D(128, 3, activation='relu', padding='same')(x)
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)
y = layers.MaxPooling2D(2, strides=2)(y)

residual = layers.Conv2D(128, 1, strides=2, padding='same')(x) ← Используется свертка  $1 \times 1$  для линейного
                                                               снижения размерности исходного тензора x,
                                                               чтобы получить форму как у тензора y

y = layers.add([y, residual]) ← Добавление остаточного тензора
                                к выходным признакам
```

## НЕДОСТАТОЧНАЯ РЕПРЕЗЕНТАТИВНОСТЬ В ГЛУБОКОМ ОБУЧЕНИИ

В модели Sequential каждый последующий слой представления строится на основе предыдущего, то есть он имеет доступ только к информации, содержащейся в активации предыдущего слоя. Если один из слоев будет слишком маленьким (например, имеет признаки со слишком низкой размерностью), тогда модель будет ограничена объемом информации, содержащимся в активациях этого слоя.

Чтобы вам было понятнее, проведем аналогию с механизмом обработки сигнала: представьте, что у вас имеется конвейер обработки аудиосигнала, состоящий из последовательности операций, каждая из которых принимает результат предыдущей операции. Тогда, если одна операция ограничит сигнал низкочастотным диапазоном (например, 0–15 кГц), последующие операции не смогут восстановить обрезанные частоты. Если информация теряется, она теряется навсегда. Остаточные связи, путем повторного внедрения более ранней информации в последующие операции, отчасти решают эту проблему.

## ЗАТУХАНИЕ ГРАДИЕНТА В ГЛУБОКОМ ОБУЧЕНИИ

Алгоритм обратного распространения ошибки — основной алгоритм, используемый для обучения глубоких нейронных сетей, — основан на распространении сигнала обратной связи от вывода к ранним слоям. Если сигналу приходится распространяться через глубокий стек слоев, он может ослабнуть или даже полностью потеряться, что сделает сеть необучаемой. Эта проблема известна как *затухание градиентов*.

Эта проблема проявляется и в глубоких, и в рекуррентных сетях с очень длинными последовательностями — в обоих случаях сигналу обратной связи приходится распространяться через длинные последовательности операций. Вы уже знакомы с решением этой проблемы, которое используется уровнем LSTM в рекуррентных сетях: оно предполагает внедрение *несущего потока*, распространяющего информацию параллельно главному потоку обработки. Остаточные связи помогают добиться аналогичного эффекта в глубоких сетях прямого распространения, но они еще проще: они внедряют простой линейный несущий поток параллельно главному стеку слоев, и это помогает распространить градиенты через сколь угодно глубокие стеки слоев.

### 7.1.5. Повторное использование экземпляров слоев

Еще одной важной особенностью функционального API является возможность повторного использования экземпляра слоя. Когда вы дважды вызываете экземпляр слоя, вместо создания нового слоя в каждом вызове повторно будут использоваться те же самые веса. Это позволяет создавать модели с общими ветвями, когда имеется несколько ветвей, совместно использующих общие знания и выполняющих одинаковые операции. Другими словами, они вместе используют общие представления и совместно обучаются на разных входных наборах.

Например, рассмотрим модель, которая попытается оценить семантическое сходство двух предложений. Модель имеет два входа (два сравниваемых предложения) и выводит оценку в диапазоне между 0 и 1, где 0 означает полное отсутствие сходства между предложениями, а 1 — полную смысловую идентичность. Такая модель могла бы найти массу применений, включая устранение избыточных запросов на естественном языке в диалоговых системах.

В такой конфигурации два входных предложения взаимозаменяемы, потому что семантическое сходство является симметричным отношением: сходство А с Б идентично сходству Б с А. По этой причине нецелесообразно обучать две независимые модели для обработки каждого входного предложения. Предпочтительнее было бы обрабатывать оба одним слоем LSTM. Представления этого слоя LSTM (его веса) определяются на основе обоих входов одновременно. Мы называем это *сиамской моделью LSTM*, или *общим LSTM*.

Вот как можно реализовать такую модель с использованием приема совместного (или повторного) использования слоя в функциональном API фреймворка Keras:

```
from keras import layers
from keras import Input
from keras.models import Model
lstm = layers.LSTM(32)
left_input = Input(shape=(None, 128))
left_output = lstm(left_input)

    ↓
    | Создание
    | единственного
    | экземпляра слоя
    | LSTM, выполняется
    | однократно
    |
    | Конструирование
    | левой ветви модели:
    | на вход подаются
    | последовательности
    | переменной
    | длины векторов
    | с размерностью 128
```

```

right_input = Input(shape=(None, 128))
right_output = lstm(right_input)

merged = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)

model = Model([left_input, right_input], predictions)
model.fit([left_data, right_data], targets)

```

**Добавление классификатора сверху**

**Создание и обучение модели: в процессе обучения такой модели веса слоя LSTM обновляются в процессе обработки обоих входов**

**Конструирование правой ветви модели: обращаясь к существующему экземпляру слоя, вы повторно используете его веса**

Естественно, экземпляр слоя можно использовать неоднократно — любое количество раз, при этом повторно будут использоваться одни и те же веса.

## 7.1.6. Модели как слои

Важно отметить, что функциональный API позволяет использовать модели как слои — фактически о моделях можно думать как о «больших слоях». Это верно для обоих классов — `Sequential` и `Model`. Другими словами, можно вызвать модель, передав ей входной тензор, и получить выходной тензор:

```
y = model(x)
```

Если модель принимает несколько входных тензоров и возвращает несколько выходных тензоров, ее следует вызывать со списками тензоров:

```
y1, y2 = model([x1, x2])
```

Вызывая экземпляр модели, вы повторно используете ее веса — в точности как при вызове экземпляра слоя. Вызов любого экземпляра — слоя или модели — всегда влечет за собой повторное использование существующего полученного представления экземпляра, что совершенно понятно.

Простым примером практического применения повторного использования экземпляра модели может служить модель зрения, которая в качестве входа использует сдвоенную камеру: две параллельные камеры, отстоящие друг от друга на пару сантиметров (один дюйм). Такая модель может воспринимать глубину, что может пригодиться во многих приложениях. Вам не нужно создавать две независимые модели для извлечения визуальных признаков из изображений, передаваемых левой и правой камерами, перед объединением двух потоков. Низкоуровневую обработку двух входных потоков можно выполнять сообща, то есть задействовать слои, совместно использующие одни и те же веса и, соответственно, представления. Вот как можно реализовать сиамскую модель зрения (с общей сверточной основой) в Keras:

```

from keras import layers
from keras import applications
from keras import Input
Базовая модель обработки
изображения — сеть Xception
(только сверточная основа)

xception_base = applications.Xception(weights=None,
                                         include_top=False) ←

left_input = Input(shape=(250, 250, 3)) | На вход подаются изображения
right_input = Input(shape=(250, 250, 3)) | в формате RGB и с размером 250 × 250

left_features = xception_base(left_input)
right_input = xception_base(right_input) | Одна и та же модель вызывается дважды

merged_features = layers.concatenate(
    [left_features, right_input], axis=-1) ←
                                                | Объединенный набор признаков
                                                | содержит информацию из правого
                                                | и левого источников визуальной
                                                | информации

```

### 7.1.7. Подведение итогов

На этом мы завершаем введение в функциональный API фреймворка Keras — основной инструмент для создания глубоких нейронных сетей с продвинутыми архитектурами. Теперь вы знаете:

- ❑ что можно использовать помимо `Sequential` API, когда потребуется нечто более мощное, чем линейный стек слоев;
- ❑ как средствами функционального API фреймворка Keras создавать модели с несколькими входами, несколькими выходами и сложной внутренней топологией;
- ❑ как повторно использовать экземпляры слоев или моделей в разных обрабатывающих ветвях, вызывая один и тот же экземпляр слоя или модели несколько раз.

## 7.2. Исследование и мониторинг моделей глубокого обучения с использованием обратных вызовов Keras и TensorBoard

В этом разделе мы рассмотрим способы получения более полного доступа к внутренним механизмам модели во время обучения и управления ими. Запуск процедуры обучения на большом наборе данных и продолжительностью в десятки эпох вызовом `model.fit()` или `model.fit_generator()` напоминает запуск бумажного самолетика: придав начальный импульс, вы больше никак не управляете ни траекторией его полета, ни местом приземления. Чтобы избежать отрицательных результатов (и потери бумажного самолетика), лучше использовать не бумажный самолетик, а управляемый беспилотник, анализирующий окружающую обстановку,

посылающий информацию о ней обратно оператору и автоматически управляющий рулями в зависимости от своего текущего состояния. Приемы, которые будут представлены здесь, превратят вызов `model.fit()` из бумажного самолетика в интеллектуальный автономный беспилотник, способный оценивать свое состояние и своевременно выполнять управляющие воздействия.

### 7.2.1. Применение обратных вызовов для воздействия на модель в ходе обучения

Многие аспекты обучения модели нельзя предсказать заранее. Например, нельзя предсказать заранее количество эпох, обеспечивающее оптимальное значение потерь на проверочном наборе. В примерах, приводившихся до сих пор, использовалась стратегия обучения с достаточно большим количеством эпох. Таким образом достигался эффект переобучения, когда сначала выполняется первый прогон, чтобы выяснить необходимое количество эпох обучения, а затем второй — новый, с самого начала — с выбранным оптимальным количеством эпох. Конечно, это довольно расточительная стратегия.

Гораздо лучше было бы остановить обучение, как только выяснится, что оценка потерь на проверочном наборе перестала улучшаться. Это можно реализовать с использованием механизма обратных вызовов в Keras. *Обратный вызов* — это объект (экземпляр класса, реализующего конкретные методы), который передается в модель через вызов `fit` и который будет вызываться моделью в разные моменты в процессе обучения. Он имеет доступ ко всей информации о состоянии модели и ее качестве и может предпринимать следующие действия: прерывать обучение, сохранять модели, загружать разные наборы весов или как-то иначе изменять состояние модели.

Вот несколько примеров использования обратных вызовов:

- ❑ *фиксация состояния модели в контрольных точках* — сохранение текущих весов модели в разные моменты в ходе обучения;
- ❑ *ранняя остановка* — прерывание обучения, когда оценка потерь на проверочных данных перестает улучшаться (и, конечно, сохранение лучшего варианта модели, полученного в ходе обучения);
- ❑ *динамическая корректировка значений некоторых параметров в процессе обучения*, например шага обучения оптимизатора;
- ❑ *журналирование оценок для обучающего и проверочного наборов данных в ходе обучения или визуализация представлений, получаемых моделью, по мере их обновления* — индикатор выполнения в Keras, с которым вы уже знакомы, — обратный вызов!

Модуль `keras.callbacks` включает в себя ряд встроенных обратных вызовов. Вот далеко не полный список:

```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Рассмотрим некоторые из них, чтобы получить представление о том, как ими пользоваться: `ModelCheckpoint`, `EarlyStopping` и `ReduceLROnPlateau`.

## Обратные вызовы `ModelCheckpoint` и `EarlyStopping`

Обратный вызов `EarlyStopping` можно использовать для прерывания процесса обучения, если находящаяся под наблюдением целевая метрика не улучшалась на протяжении заданного количества эпох. Например, этот обратный вызов позволит прервать обучение после наступления эффекта переобучения и тем самым избежать повторного обучения модели в течение меньшего количества эпох. Этот обратный вызов обычно используется в комбинации с `ModelCheckpoint`, который позволяет сохранять состояние модели в ходе обучения (и, при необходимости, сохранять только лучшую модель: версию модели, достигшую лучшего качества к концу эпохи):

```
Передача обратных вызовов в модель
через аргумент callbacks метода fit в виде
списка. Вы можете передать любое
количество обратных вызовов

import keras

callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor='val_acc',
        patience=1,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath='my_model.h5',
        monitor='val_loss',
        save_best_only=True,
    )
]

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

model.fit(x, y,
          epochs=10,
          batch_size=32,
          callbacks=callbacks_list,
          validation_data=(x_val, y_val))
```

Прерывание обучения, когда качество модели перестает улучшаться

Следение за изменением точности модели на проверочных данных

Прерывание обучения, когда точность не улучшается дольше чем в течение одной эпохи (другими словами, в течение двух эпох)

Сохранение текущих весов после каждой эпохи

Путь к файлу модели

Эти аргументы требуют, чтобы файл модели не затирался, если значение `val_loss` не улучшилось, что позволяет сохранять только лучшую модель

Мы следим за точностью, поэтому она должна быть частью набора метрик модели

Обратите внимание: поскольку обратный вызов следит за потерями и точностью на проверочных данных, мы должны передать `validation_data` в вызов `fit`

## Обратный вызов ReduceLROnPlateau

Этот обратный вызов можно использовать для снижения скорости обучения, когда потери на проверочных данных перестают уменьшаться. Уменьшение или увеличение скорости обучения в точке перегиба кривой потерь — эффективная стратегия выхода из локального минимума в ходе обучения. Следующий пример демонстрирует применение обратного вызова `ReduceLROnPlateau`:

```
callbacks_list = [
    keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss' ← Мониторинг потерь модели на проверочных данных
        factor=0.1, ← Уменьшение скорости обучения в 10 раз
        patience=10, ← Произведение вызова, когда потери на проверочных
    )                               данных не улучшаются в течение 10 эпох
]
]

model.fit(x, y,
           epochs=10,
           batch_size=32,
           callbacks=callbacks_list,
           validation_data=(x_val, y_val))
```

Поскольку обратный вызов следует за потерями на проверочных данных, мы должны передать `validation_data` в вызов `fit`

## Разработка своего обратного вызова

Если в ходе обучения потребуется выполнить какие-то особые действия, не предусмотренные ни одним из встроенных обратных вызовов, можно написать свой обратный вызов. Обратные вызовы реализуются путем наследования класса `keras.callbacks.Callback`. Вы можете реализовать любые из следующих методов с говорящими именами, которые будут вызываться в соответствующие моменты в ходе обучения:

<code>on_epoch_begin</code>	← Вызывается в начале каждой эпохи
<code>on_epoch_end</code>	← Вызывается в конце каждой эпохи
<code>on_batch_begin</code>	← Вызывается перед началом обработки каждого пакета
<code>on_batch_end</code>	← Вызывается сразу после окончания обработки каждого пакета
<code>on_train_begin</code>	← Вызывается в начале обучения
<code>on_train_end</code>	← Вызывается в конце обучения

Все эти методы вызываются с аргументом `logs` — словарем, содержащим информацию о предыдущем пакете, эпохе или цикле обучения: метрики обучения и проверки и т. д. Кроме того, обратный вызов имеет доступ к следующим атрибутам:

- `self.model` — экземпляр модели, вызвавшей этот обратный вызов;
- `self.validation_data` — значение, переданное методу `fit` в качестве проверочных данных.

Вот простой пример нестандартного обратного вызова, который сохраняет на диск (как массивы NumPy) активации всех слоев модели после окончания каждой эпохи, вычисленные по первому образцу в проверочном наборе:

```
import keras
import numpy as np

class ActivationLogger(keras.callbacks.Callback):
    def set_model(self, model):
        self.model = model
        layer_outputs = [layer.output for layer in model.layers]
        self.activations_model = keras.models.Model(model.input,
                                                      layer_outputs)

    def on_epoch_end(self, epoch, logs=None):
        if self.validation_data is None:
            raise RuntimeError('Requires validation_data.')
        validation_sample = self.validation_data[0][0:1]
        activations = self.activations_model.predict(validation_sample)
        f = open('activations_at_epoch_' + str(epoch) + '.npz', 'wb')
        np.savez(f, activations)
        f.close()

    Получение первого образца
    из проверочных данных
```

Вызывается родительской моделью перед обучением, чтобы сообщить обратному вызову, какая модель будет обращаться к нему

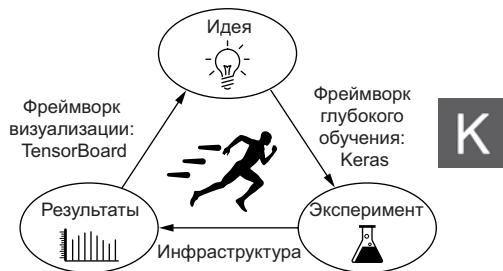
Экземпляр модели, которая будет возвращать активации для всех слоев

Сохранение массива на диск

Это все, что нужно знать об обратных вызовах, все остальное — технические детали, информацию о которых вы легко сможете отыскать самостоятельно. Теперь вы сможете выполнять журналирование любой информации или оказывать управляющие воздействия на модель Keras в ходе обучения.

## 7.2.2. Введение в TensorBoard: фреймворк визуализации TensorFlow

Для плодотворных исследований или разработки хороших моделей необходимо иметь разностороннюю, часто обновляющуюся информацию о происходящем внутри модели в ходе экспериментов. Главная цель экспериментов: получить информацию о том, насколько хорошо работает модель, — как можно больше информации. Движение вперед носит итеративный, или циклический, характер: вы начинаете с идеи и разрабатываете план эксперимента, который подтвердит или опровергнет ее. Вы запускаете эксперимент и обрабатываете полученную информацию. Это дает толчок к рождению новой идеи. Чем больше итераций в этом цикле вы выполните, тем совершеннее и мощнее будут становиться ваши идеи. Keras поможет вам перейти от идеи к эксперименту в кратчайшие сроки, а быстрые GPU помогут вам получить результаты эксперимента как можно быстрее. Но как быть с обработкой результатов? В этом вам поможет TensorBoard.

**Рис. 7.9.** Циклическое движение вперед

В этом разделе мы познакомимся с TensorBoard, инструментом визуализации, основанным на использовании браузера, входящего в состав TensorFlow. Обратите внимание: его можно использовать для исследования моделей Keras, только когда в качестве низкоуровневого механизма обработки тензоров Keras использует TensorFlow.

Основное назначение TensorBoard — помочь визуально наблюдать за происходящим внутри модели в процессе обучения. Отслеживая больший объем информации, нежели просто окончательные потери модели, вы сможете получить более четкое представление о том, что делает или чего не делает модель, и быстрее добиться прогресса. TensorBoard открывает доступ к некоторым замечательным возможностям через самое обычное окно браузера:

- ❑ визуальный мониторинг метрик в ходе обучения;
- ❑ визуализация архитектуры модели;
- ❑ вывод гистограмм активаций и градиентов;
- ❑ исследование векторных представлений в трехмерном пространстве.

Рассмотрим эти возможности на простом примере. Мы обучим одномерную сверточную сеть на данных IMDB для решения задачи определения эмоциональной окраски отзывов.

Эта модель напоминает ту, что вы видели в последнем разделе главы 6. Мы рассмотрим только первые 2000 самых часто используемых слов из словаря IMDB, чтобы получить более простое визуальное изображение векторных представлений слов.

#### **Листинг 7.7.** Модель классификации текста для анализа в TensorBoard

```
import keras
from keras import layers
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 2000 ← Количество слов, рассматриваемых как признаки
max_len = 500 ← Обрезка текста после этого количества слов (в числе
                 max_features самых распространенных слов)
```

Продолжение ↗

**Листинг 7.7** (продолжение)

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)

model = keras.models.Sequential()
model.add(layers.Embedding(max_features, 128,
                           input_length=max_len,
                           name='embed'))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
```

Перед началом использования TensorBoard необходимо создать каталог, куда будут сохраняться файлы журналов, генерируемые этим инструментом.

**Листинг 7.8.** Создание каталога для файлов журналов TensorBoard

```
$ mkdir my_log_dir
```

Теперь запустим обучение, передав экземпляр TensorBoard в качестве обратного вызова. Этот обратный вызов будет записывать события на диск в указанный каталог.

**Листинг 7.9.** Обучение модели с обратным вызовом TensorBoard

```
callbacks = [
    keras.callbacks.TensorBoard(
        log_dir='my_log_dir',           ← Файлы журналов будут сохраняться в этом каталоге
        histogram_freq=1,             ← Запись гистограммы активаций в каждой эпохе
        embeddings_freq=1,            ← Запись векторных представлений в каждой эпохе
    )
]
history = model.fit(x_train, y_train,
                     epochs=20,
                     batch_size=128,
                     validation_split=0.2,
                     callbacks=callbacks)
```

После этого можно запустить сервер TensorBoard из командной строки, указав, что тот должен читать журналы, которые в настоящий момент записывает обратный вызов. Утилита `tensorboard` должна автоматически установиться вместе с фреймворком TensorFlow:

```
$ tensorboard --logdir=my_log_dir
```

После этого можно запустить браузер, перейти по адресу `http://localhost:6006` и посмотреть, как протекает процесс обучения модели (рис. 7.10). Помимо

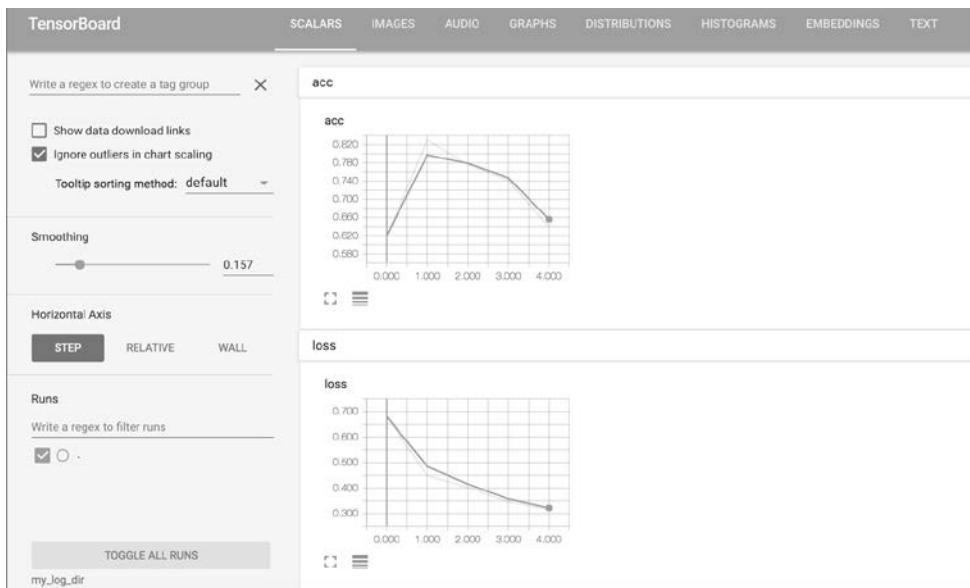


Рис. 7.10. TensorBoard: мониторинг метрик

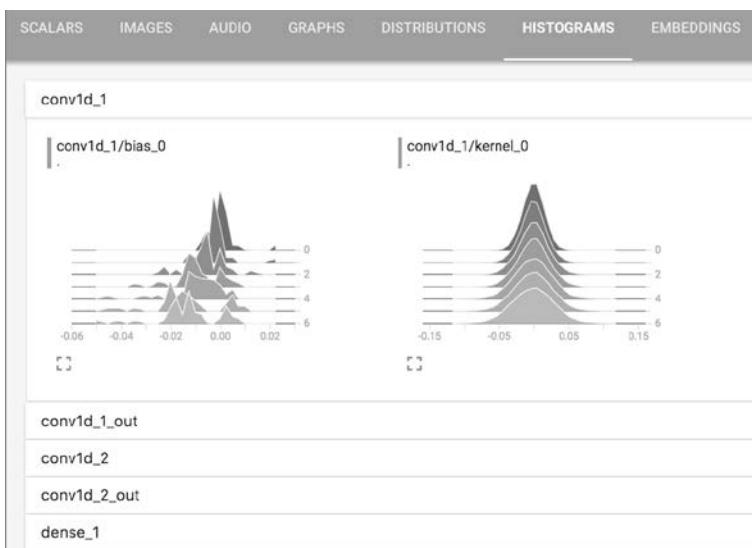
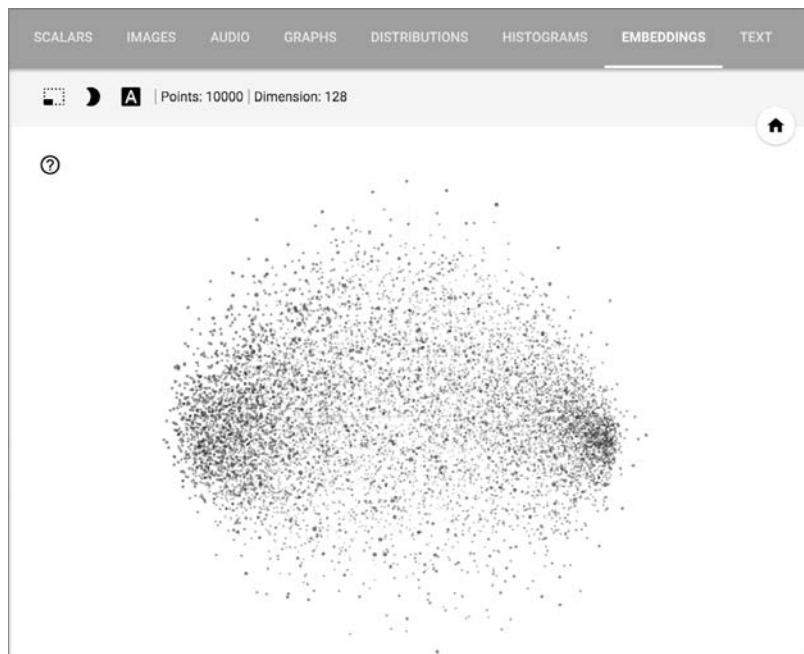


Рис. 7.11. TensorBoard: гистограммы активаций

динамически обновляющихся графиков метрик, определяемых на этапах обучения и проверки, можно, перейдя на вкладку **Histograms** (Гистограммы), найти превосходные гистограммы, отображающие значения активации, получаемые вашими слоями (рис. 7.11).

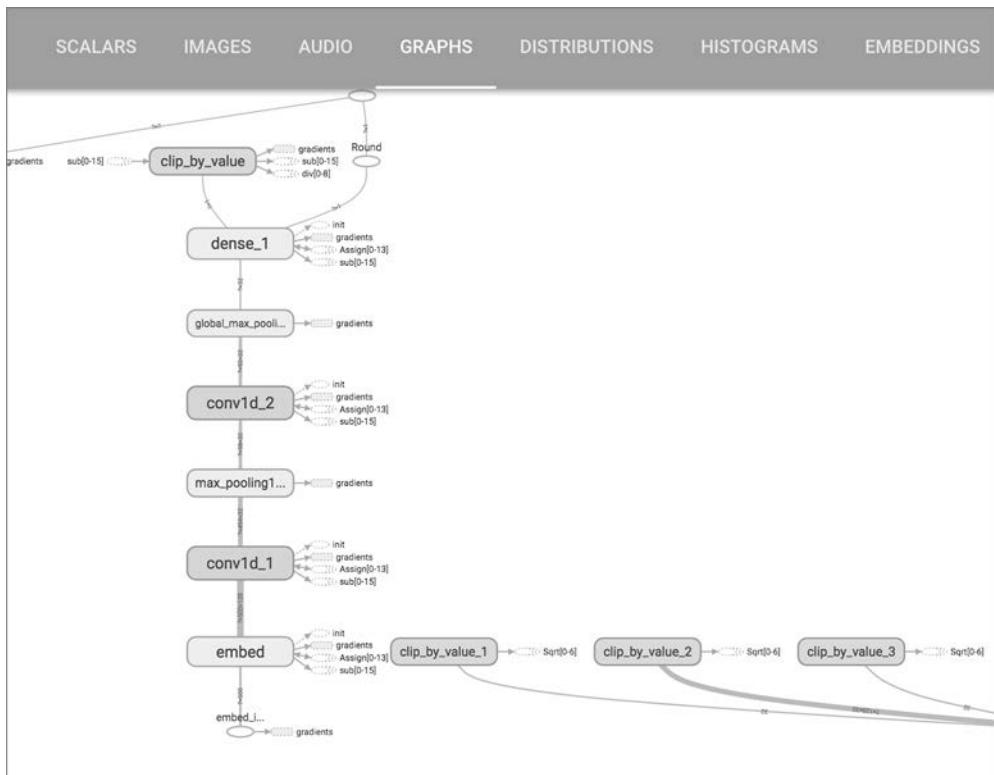
Вкладка **Embeddings** (Векторные представления) позволяет рассмотреть местоположения и пространственные отношения 10 000 слов во входном словаре, выявленные начальным слоем **Embedding**. Поскольку векторное пространство имеет 128 измерений, TensorBoard автоматически снижает его размерность до двух или трех, используя алгоритм уменьшения размерности по вашему выбору: метод главных компонент (Principal Component Analysis, PCA) или метод нелинейного снижения размерности и визуализации многомерных данных t-SNE (t-distributed Stochastic Neighbor Embedding). На рис. 7.12 изображено облако точек, в котором четко видно два кластера: слова с положительной и с отрицательной окраской. Визуальное представление помогает сразу же заметить, что векторные представления, получаемые с определенной целью, приводят к моделям, которые полностью характерны для решаемой задачи, — именно поэтому использование предварительно обученных обобщенных векторных представлений слов редко является хорошей идеей.



**Рис. 7.12.** TensorBoard: интерактивная карта трехмерного пространства векторных представлений слов

На вкладке **Graphs** (Диаграммы) изображены интерактивные диаграммы низкоХровневых операций, выполняемых фреймворком TensorFlow в ходе обучения модели Keras (рис. 7.13). Как видите, в действительности за кулисами выполняется намного больше операций, чем можно было бы ожидать. Выраженная в конструкциях Keras, только что созданная модель выглядит просто — маленький стек

из простых слоев, однако, чтобы заставить ее работать, за кулисами создается очень сложный граф. Большая часть этого графа связана с организацией градиентного спуска. Такая разница в сложности между тем, что вы видите, и тем, чем управляете, — главный мотив к использованию Keras для создания моделей вместо низкоуровневого TensorFlow. Keras делает процесс разработки моделей до смешного простым.

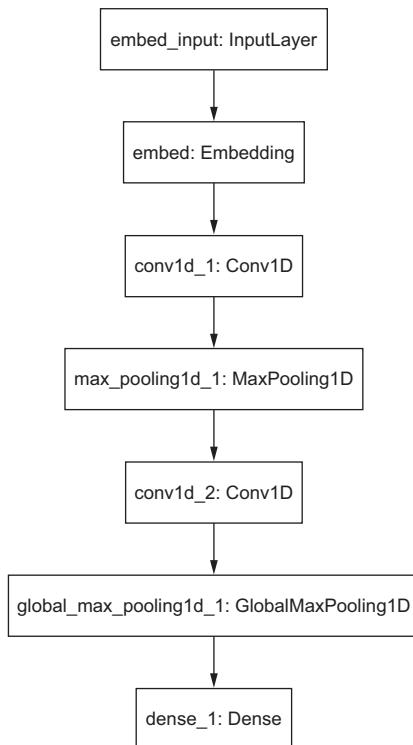


**Рис. 7.13.** TensorBoard: график операций TensorFlow

Примечательно, что Keras поддерживает также другой, более ясный способ представления моделей в виде графов слоев вместо графов операций TensorFlow: утилиту `keras.utils.plot_model`. Чтобы воспользоваться ею, нужно установить библиотеки для Python `pydot` и `pydot-ng`, а также библиотеку `graphviz`. Посмотрим, что может эта утилита:

```
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

Этот вызов создаст изображение в формате PNG, представленное на рис. 7.14.



**Рис. 7.14.** Изображение модели в виде графа слоев, созданное утилитой `plot_model`

С помощью этой утилиты также можно отобразить информацию о форме слоев в графе. Следующий пример создает изображение с топологией модели, передавая утилите `plot_model` параметр `show_shapes` (рис. 7.15):

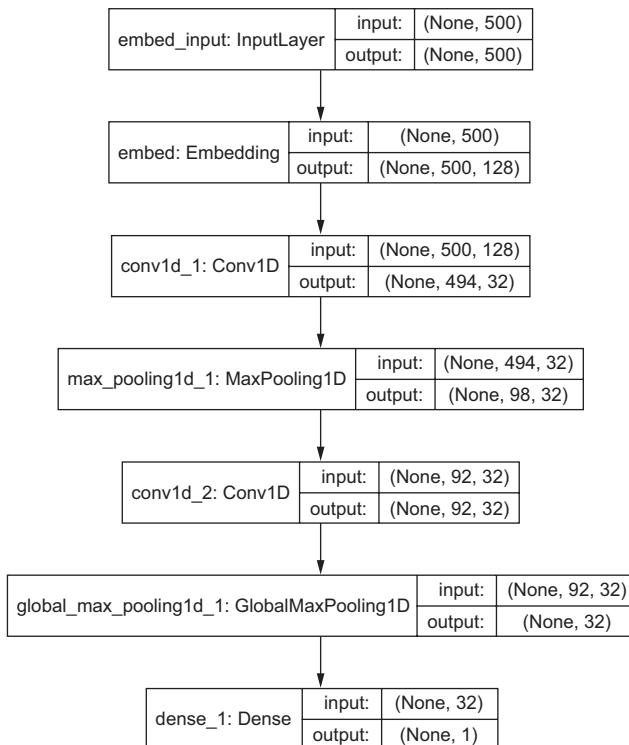
```

from keras.utils import plot_model

plot_model(model, show_shapes=True, to_file='model.png')
  
```

### 7.2.3. Подведение итогов

- ❑ Обратные вызовы Keras дают простую возможность следить за происходящим внутри модели в ходе ее обучения и автоматически предпринимать какие-либо действия, опираясь на ее состояние.
- ❑ Если в качестве низкоуровневого механизма обработки тензоров используется TensorFlow, появляется возможность использовать TensorBoard — великолепный инструмент визуализации процессов, протекающих в модели, в окне браузера. Для его использования с моделями Keras нужно добавлять в модели обратный вызов `TensorBoard`.



**Рис. 7.15.** Изображение модели с информацией о форме слоев

## 7.3. Извлечение максимальной пользы из моделей

Построение архитектур вслепую — неплохой подход, когда просто-напросто нужно, чтобы все работало. В этом разделе мы сделаем шаг от «просто работающих» к «продвинутым моделям, побеждающим в состязаниях по машинном обучению», и рассмотрим некоторые приемы создания современных моделей глубокого обучения.

### 7.3.1. Шаблоны улучшенных архитектур

В предыдущем разделе мы рассмотрели один из важнейших шаблонов проектирования: остаточные связи. Однако есть еще два шаблона, которые вы должны знать: нормализация и раздельные свертки по глубине (*depthwise separable convolution*). Эти шаблоны особенно хорошо подходят для создания высококачественных глубоких сверточных нейронных сетей, но их нередко можно встретить во многих других типах архитектур.

## Пакетная нормализация

*Нормализация* — это широкая категория методов, стремящихся сделать сходство разных образцов более заметным для модели машинного обучения, что помогает модели выделять и обобщать новые данные. В этой книге вы уже несколько раз видели наиболее распространенную форму нормализации: центрирование данных по нулю вычитанием среднего значения и приздание единичного стандартного отклонения делением на их стандартное отклонение. Фактически такая нормализация предполагает, что данные соответствуют нормальному закону распределения (или закону Гаусса), центрируя и приводя это распределение к единичной дисперсии:

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

В предыдущих примерах нормализация выполнялась перед передачей данных в модели. Однако нормализация должна проводиться после каждого преобразования, выполняемого сетью: даже если данные на входе в сеть `Dense` или `Conv2D` имеют среднее значение 0 и единичную дисперсию, нет оснований полагать, что то же самое можно будет сказать в отношении данных на выходе.

Пакетная нормализация — это тип слоя (`BatchNormalization` в Keras), введенный в 2015 году Сергеем Йоффе (Sergey Ioffe) и Кристианом Сегеди (Christian Szegedy)<sup>1</sup>; он может адаптивно нормализовать данные, даже если среднее и дисперсия изменяются во время обучения. Его принцип действия основан на вычислении экспоненциального скользящего среднего и дисперсии данных, наблюдаемых в процессе обучения. Основное назначение пакетной нормализации — помочь распространению градиента подобно остаточным связям и дать возможность создавать более глубокие сети. Некоторые глубокие сети могут обучаться, только если они включают в себя несколько слоев `BatchNormalization`. Например, слои `BatchNormalization` широко используются во многих продвинутых архитектурах сверточных нейронных сетей, входящих в состав Keras, таких как ResNet50, Inception V3 и Xception.

Обычно слой `BatchNormalization` используется после сверточного или полно связного слоя:

```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))  
conv_model.add(layers.BatchNormalization()) ← После слоя Conv  
  
dense_model.add(layers.Dense(32, activation='relu'))  
dense_model.add(layers.BatchNormalization()) ← После слоя Dense
```

Слой `BatchNormalization` принимает аргумент `axis`, определяющий ось признаков для нормализации. По умолчанию этот аргумент принимает значение `-1`,

---

<sup>1</sup> Sergey Ioffe and Christian Szegedy, «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», *Proceedings of the 32nd International Conference on Machine Learning* (2015), <https://arxiv.org/abs/1502.03167>.

что соответствует последней оси во входном тензоре. Это правильное значение, когда используются слои Dense, Conv1D, рекуррентные слои и слои Conv2D со значением "channels\_last" в аргументе `data_format`. Но в слоях Conv2D со значением "channels\_first" в аргументе `data_format` ось признаков — это ось с индексом 1; поэтому в таких случаях слою `BatchNormalization` следует передавать число 1 аргументе `axis`.

## ПАКЕТНАЯ РЕНОРМАЛИЗАЦИЯ

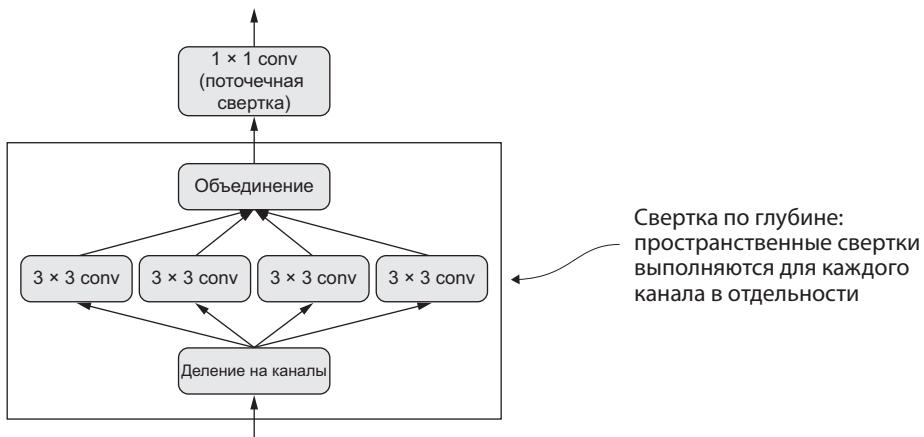
Не так давно, в 2017 году, Сергеем Йоффе был предложен более совершенный метод, нежели обычная пакетная нормализация, — метод *пакетной ренормализации*<sup>1</sup>. Он предлагает очевидные преимущества по сравнению с пакетной нормализацией, без дополнительных накладных расходов. На данный момент еще рано говорить, вытеснит ли он пакетную нормализацию, но я думаю, что это вполне возможно. А совсем недавно Гюнтер Кламбаер (Günter Klambauer) с коллегами представили *самонормализующиеся нейронные сети*<sup>2</sup>, которые позволяют сохранять данные в нормализованном состоянии после прохождения через любой слой Dense за счет использования специальных функций активации (`selu`) и инициализации (`leculn_normal`). Эта схема выглядит довольно интересной, но пока она ограничивается только полносвязными сетями и ее полезность на данный момент не подтверждена широкими исследованиями.

## Раздельные свертки по глубине

Что бы вы подумали, если бы я сказал, что существует такой слой, который можно использовать взамен Conv2D и с помощью которого сделать модель более легкой (с меньшим количеством обучаемых весовых параметров) и быстрой (с меньшим количеством операций с вещественными числами), а также повысить качество решения задачи на несколько процентных пунктов? Всеми перечисленными качествами обладает слой раздельной свертки по глубине (`SeparableConv2D`). Этот слой выполняет пространственную свертку каждого канала во входных данных в отдельности перед смешиванием выходных каналов посредством поточечной свертки (свертки  $1 \times 1$ ), как показано на рис. 7.16. Это эквивалентно раздельному выделению пространственных и канальных признаков, что оправданно, если предполагается сильная корреляция пространственных местоположений на входе, но разные каналы практически не зависят друг от друга. Он требует намного меньше параметров и выполняет меньше вычислений, благодаря чему получаются более быстрые модели с меньшими размерами. И поскольку это более препрезентативно эффективный способ выполнения свертки, он позволяет получать более качественные представления с меньшим объемом исходных данных и, соответственно, более качественные модели.

<sup>1</sup> Sergey Ioffe, «Batch Renormalization: Towards Reducing Minibatch Dependence in BatchNormalized Models» (2017), <https://arxiv.org/abs/1702.03275>.

<sup>2</sup> Günter Klambauer et al., «Self-Normalizing Neural Networks», Conference on Neural Information Processing Systems (2017), <https://arxiv.org/abs/1706.02515>.



**Рис. 7.16.** Раздельная свертка по глубине: за сверткой по глубине следует поточечная свертка

Эти преимущества особенно важны для обучения небольших моделей с нуля на ограниченном наборе данных. Например, вот как можно построить легковесную нейронную сеть со сверткой по глубине для решения задачи классификации изображений (с применением классификатора `softmax`) при небольшом объеме обучающих данных:

```
from keras.models import Sequential, Model
from keras import layers

height = 64
width = 64
channels = 3
num_classes = 10

model = Sequential()
model.add(layers.SeparableConv2D(32, 3,
                                activation='relu',
                                input_shape=(height, width, channels,)))
model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.GlobalAveragePooling2D())

model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

В отношении крупных моделей раздельные свертки по глубине составляют основу архитектуры Xception высококачественных сверточных нейронных сетей, входящей в состав Keras. Узнать больше о теоретических основах раздельной свертки по глубине и архитектуре Xception можно в моей статье «Xception: Deep Learning with Depthwise Separable Convolutions»<sup>1</sup>.

### 7.3.2. Оптимизация гиперпараметров

При создании модели глубокого обучения приходится принимать множество решений, кажущихся произвольными: сколько слоев включить в стек? сколько параметров или фильтров должно быть в каждом слое? использовать ли функцию активации `relu` или какую-то другую? использовать ли `BatchNormalization` после данного слоя? какой шаг прореживания выбрать? И так далее. Все эти архитектурные параметры называют *гиперпараметрами*, чтобы отличать их от параметров модели, которые обучаются посредством обратного распространения ошибки.

На практике инженеры и исследователи, занимающиеся машинным обучением, со временем накапливают опыт, который помогает им делать правильный выбор, — они обретают навыки настройки гиперпараметров. Однако формальных правил не существует. Чтобы дойти до самого предела возможностей в решении какой-либо задачи, нельзя довольствоваться произвольным выбором, сделанным по ошибке. Ваши первоначальные решения часто будут не самыми оптимальными, даже если вы обладаете хорошей интуицией. Вы можете менять свой выбор, выполняя настройки вручную и повторно обучая модель, — именно этим большую часть времени занимаются инженеры и исследователи машинного обучения. Однако перебор разных параметров не должен быть вашей основной работой — это дело лучше доверить машине.

Другими словами, вам нужно автоматически, систематически и принципиально исследовать пространство возможных решений. Вам нужно эмпирически исследовать пространство архитектур и найти ту из них, которая сможет обеспечить лучшее качество. Именно эту задачу решает автоматическая оптимизация гиперпараметров: это огромная и очень важная область исследований.

Вот как выглядит типичный процесс оптимизации гиперпараметров:

1. Выбрать набор гиперпараметров (автоматически).
2. Создать соответствующую модель.
3. Обучить ее на обучающих данных и оценить качество на проверочных данных.
4. Выбрать следующий набор гиперпараметров (автоматически).
5. Повторить.
6. Получить окончательную оценку качества на контрольных данных.

<sup>1</sup> François Chollet, «Xception: Deep Learning with Depthwise Separable Convolutions», Conference on Computer Vision and Pattern Recognition (2017), <https://arxiv.org/abs/1610.02357>.

Большое значение в этом процессе имеет алгоритм, использующий историю оценок качества на проверочных данных для разных наборов гиперпараметров, с тем чтобы выбрать следующий набор гиперпараметров. Существует множество возможных претендентов на эту роль: байесовская оптимизация, генетические алгоритмы, простой случайный поиск и т. д.

Обучение весов модели выполняется относительно просто: вычисляется функция потерь на мини-пакете данных и затем используется алгоритм обратного распространения ошибки для смещения весов в нужном направлении. Изменение гиперпараметров, напротив, очень сложная задача, особенно если принять во внимание следующее:

- ❑ Вычисление сигнала обратной связи (действительно ли данный набор гиперпараметров ведет к увеличению качества модели для данной задачи?) может обходиться очень дорого: для этого нужно создать и обучить новую модель с нуля.
- ❑ Пространство гиперпараметров обычно состоит из дискретных решений и поэтому не является непрерывным и дифференцируемым. Как следствие, метод градиентного спуска нельзя применить в пространстве гиперпараметров. Вместо этого приходится полагаться на другие приемы оптимизации, не такие эффективные, как метод градиентного спуска.

Проблемы сложны, а область еще молода, поэтому в настоящее время в нашем распоряжении имеется весьма ограниченный набор инструментов для оптимизации моделей. Часто случайный поиск (многократный выбор случайных значений гиперпараметров) оказывается лучшим решением, несмотря на то что он самый простой. Однако я обнаружил один инструмент, который достоверно лучше случайного поиска, — Hyperopt (<https://github.com/hyperopt/hyperopt>), библиотеку на Python для оптимизации гиперпараметров: внутренне она использует деревья оценок Парзена для предсказания наборов гиперпараметров, которые с высокой вероятностью дадут положительный результат. Еще одна библиотека, с названием Hyperas (<https://github.com/maxrimperla/hyperas>), интегрирует Hyperopt для использования с моделями Keras. Обязательно обратите на нее внимание.

### ПРИМЕЧАНИЕ

При использовании автоматического оптимизатора гиперпараметров важно помнить об одной важной проблеме — переобучении на проверочном наборе. Поскольку настройка гиперпараметров выполняется по результатам оценки на проверочных данных, фактически происходит их обучение на проверочном наборе, и, как следствие, быстро наступает эффект их переобучения. Всегда помните об этом.

В целом оптимизация гиперпараметров — мощный метод, абсолютно необходимый в любых задачах для создания моделей, способных победить в состязаниях по машинному обучению. Когда-то люди вручную выбирали признаки, которые затем передавались в поверхностные модели машинного обучения. Этот подход был очень неоптимальен. Теперь глубокое обучение автоматизирует конструирование

иерархических признаков — признаков, выделяемых с использованием сигнала обратной связи, а не вручную, как и должно быть. Точно так же вы не должны вручную настраивать архитектуру своих моделей; вы должны оптимизировать их на принципиальной основе. На момент написания этих строк область автоматической оптимизации гиперпараметров была еще очень юной и незрелой, как само глубокое обучение несколько лет тому назад, но я полагаю, что в ближайшие годы ситуация кардинально изменится.

### 7.3.3. Ансамблирование моделей

Еще один метод улучшения результатов в решении задач — *ансамблирование моделей*. Суть метода ансамблирования заключается в объединении прогнозов, полученных набором разных моделей, для получения лучшего прогноза. Если рассмотреть результаты соревнований по машинному обучению, например, на сайте Kaggle, можно увидеть, что победители используют очень большие ансамбли моделей, которые неизменно побеждают любые одиночные модели, даже самые лучшие.

Ансамблирование основано на предположении о том, что разные хорошие модели, обученные независимо, могут быть хороши по разным причинам: каждая модель рассматривает немного другие аспекты данных, чтобы сделать прогноз, и видит только часть «истины». Возможно, вы знакомы с древней притчей о слоне и незрячих мудрецах: группа незрячих мудрецов впервые встречает слона и, чтобы понять, что такое слон, ощупывает его. Каждый касается только одной его части, такой как туловище или нога. Затем каждый мудрец описывает свое представление о слоне: «он гибкий, как змея», «он как колонна или ствол дерева» и т. д. Незрячие мудрецы в этой притче подобны моделям машинного обучения, когда те пытаются понять многообразие обучающих данных, каждая со своей точки зрения и исходя из своих предположений (определенных уникальной архитектурой модели и случайными значениями весов, полученных в момент инициализации). Каждая видит только часть целого. Объединив точки зрения, можно получить гораздо более точное описание данных. Слон — это комбинация его частей: ни один незрячий мудрец не обладает всей истиной, но вместе они могут дать очень точное описание.

Возьмем в качестве примера задачу классификации. Самый простой способ объединить прогнозы из множества классификаторов (ансамблировать классификаторы) — получить среднее их прогнозов на этапе вывода:

```
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)

final_preds = 0.25 * (preds_a + preds_b + preds_c + preds_d) ←
    | Применение четырех разных моделей
    | для вычисления начальных прогнозов
    |
    | Этот новый массив прогнозов должен получиться
    | более точным, чем любой из начальных
```

Этот прием даст положительные результаты, только если исходные классификаторы примерно одинаково хороши. Если один будет значительно хуже других, окончательный прогноз может получиться хуже прогноза лучшего классификатора в группе.

Более эффективный способ ансамблирования классификаторов — вычисление взвешенного среднего с определением весов по проверочным данным, когда лучший классификатор получает больший вес, а худший — меньший. Для поиска оптимальных весов в ансамбле можно использовать алгоритм случайного поиска или простой оптимизации, такой как Nelder-Mead:

```
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)

final_preds = 0.5 * preds_a + 0.25 * preds_b + 0.1 * preds_c + 0.15 * preds_d
```

Эти веса (0.5, 0.25, 0.1, 0.15),  
 как предполагается, получены  
 эмпирическим путем

Существует много возможных вариантов: вы можете вычислить среднее экспоненциальное прогнозов, for instance. В общем случае простое взвешенное среднее с весами, оптимизированными на проверочных данных, может служить очень неплохим базовым решением.

Ключом к достижению успеха в результате ансамблирования является разнообразие набора классификаторов. Разнообразие — это сила. Если бы все незрячие мудрецы ощупали только хобот слона, они согласились бы, что слоны похожи на змей, и навсегда бы остались в неведении о действительной форме слона. Разнообразие — вот что обеспечивает успех ансамблирования. Выражаясь языком машинного обучения: если все ваши модели будут одинаково предвзятыми, ваш ансамбль сохранит эту предвзятость. Если ваши модели будут *предвзяты по-разному*, предвзятости будут нивелировать друг друга, и ансамбль получится более точным и надежным.

По этой причине объединяться в ансамбли должны *максимально хорошие и разные модели*. Это обычно означает использование разных архитектур или даже разных подходов к машинному обучению. Единственное, пожалуй, чего не следует делать — ансамблировать ту же сеть, обученную несколько раз, даже при разных начальных случайных значениях. Если ваши модели различаются только начальными значениями и тем, в каком порядке они обрабатывали обучающие данные, ваш ансамбль будет иметь низкое разнообразие и обеспечит лишь незначительное улучшение по сравнению с единственной моделью.

В своей практике я обнаружил один прием, дающий хорошие результаты, — однако он не является универсальным и подходит не для всякой предметной области — использование ансамбля древовидных методов (таких, как случайные леса или деревья градиентного роста) и глубоких нейронных сетей. В 2014 году

Андрей Колев (Andrei Kolev) и я вместе заняли четвертое место среди решений задачи обнаружения бозона Хиггса на сайте Kaggle ([www.kaggle.com/c/higgs-boson](http://www.kaggle.com/c/higgs-boson)), использовав ансамбль разных древовидных моделей и глубоких нейронных сетей. Примечательно, что одна из моделей в ансамбле реализовала другой метод (это был регуляризованный жадный лес — regularized greedy forest) и имела существенно худшую оценку, нежели остальные. Неудивительно, что она получила маленький вес в ансамбле. Тем не менее, к нашему удивлению, оказалось, что она значительно улучшала ансамбль в целом, потому что сильно отличалась от всех других моделей: она сохраняла информацию, к которой другие модели не имели доступа. Именно в этом заключается суть ансамблирования. Главное не то, насколько хороша ваша лучшая модель, а то, насколько разнообразны модели-кандидаты.

В последнее время большим успехом на практике пользовалась обширная категория моделей, сочетающих глубокое и поверхностное обучение. Такие модели состоят из совместно обучаемых глубокой нейронной сети и большой линейной модели. Совместное обучение семейства разных моделей — еще один способ ансамблирования.

### 7.3.4. Подведение итогов

- Для создания высококачественных глубоких сверточных нейронных сетей вам понадобятся остаточные связи, пакетная нормализация и раздельные свертки по глубине. В будущем вполне возможно, что раздельные свертки по глубине полностью вытеснят обычные свертки в одно-, двух- и трехмерных применениях благодаря их высокой эффективности.
- При создании глубоких сетей часто приходится подбирать гиперпараметры и архитектуры, которые вместе определяют качество вашей модели. Вместо того чтобы основываться на интуиции или случайных вариантах, намного лучше организовать систематический поиск в пространстве гиперпараметров, чтобы найти оптимальный вариант. На данный момент этот процесс очень затратный, а инструменты не очень хороши. Однако вам на помощь могут прийти библиотеки Hyperopt и Hyperas. Оптимизируя гиперпараметры, не забывайте о переобучении на поверочном наборе!
- Одержать победу в соревнованиях по машинному обучению и вообще получить максимально хорошие результаты в решении задач можно только с использованием больших ансамблей моделей. Ансамблирование с использованием взвешенного среднего обычно дает неплохие результаты. Однако помните: сила в разнообразии. Бессмысленно ансамблировать схожие модели; лучшие ансамбли получаются из максимально разнородных моделей (при этом, естественно, имеющих максимальную прогнозирующую способность).

## Краткие итоги главы

- В этой главе вы узнали:
  - как конструировать модели в виде произвольного графа слоев, повторно использовать слои (веса слоев) и использовать модели подобно функциям Python (как шаблоны моделей);
  - как использовать обратные вызовы Keras для наблюдения за состоянием модели в ходе обучения и выполнении действий в зависимости от этого состояния;
  - как с помощью TensorBoard визуализировать метрики, гистограммы активаций и даже пространства векторных представлений;
  - что такое пакетная нормализация, раздельная свертка по глубине и остаточные связи;
  - почему необходимо использовать оптимизацию параметров и ансамблирование моделей.
- Получив эти новые инструменты, вы теперь лучше готовы к практическому применению глубокого обучения и сможете создавать конкурентоспособные модели глубокого обучения.

# 8

# Генеративное глубокое обучение

Эта глава охватывает следующие темы:

- ✓ генерирование текста с помощью LSTM;
- ✓ реализация DeepDream;
- ✓ нейронная передача стиля;
- ✓ вариационные автокодировщики;
- ✓ генеративно-состязательные сети.

Потенциал искусственного интеллекта в подражании человеческим мыслительным процессам простирается далеко за рамки распознавания объектов и многих реактивных задач, таких как управление автомобилем. Он охватывает также творческую деятельность. Когда я впервые заявил, что в недалеком будущем большая часть художественного и культурного контента, который мы потребляем, будет создаваться со значительной помощью ИИ, я столкнулся с полным недоверием, даже со стороны тех, кто давно практикует применение методов машинного обучения. Это было в 2014 году. Спустя всего три года это недоверие исчезло. Летом 2015 года мы развлекались с алгоритмом Google DeepDream, превращавшим изображения в психodelическую мешанину из собачьих глаз и парейдолических артефактов; в 2016 году мы использовали приложение Prisma для превращения фотографий в картины разных стилей. Летом 2016 года вышел экспериментальный короткометражный фильм *Sunspring*, снятый по сценарию, написанному алгоритмом долгой краткосрочной памяти (Long Short-Term Memory, LSTM), включая диалоги. Возможно, недавно вам доводилось слушать музыку, сочиненную нейронной сетью.

Конечно, художественные произведения, созданные ИИ, которые мы видели, пока довольно низкого качества. Искусственный интеллект пока не может состязаться с людьми, сценаристами, художниками и композиторами. Впрочем, замена чело-

века никогда не была главной целью: ИИ предполагался не для замены нашего интеллекта, а для вовлечения интеллекта в нашу жизнь и работу — интеллекта другого рода. Во многих областях, и особенно в творчестве, ИИ будет использоваться людьми как инструмент для расширения своих возможностей: более широких, чем возможности ИИ.

Художественное творчество в значительной мере заключается в распознавании образов и технических навыках. Многим именно эта часть процесса кажется мало-привлекательной, а иногда даже отталкивающей. Помочь исправить эту проблему может ИИ. Наши перцептивные модальности, наш язык и наше творчество имеют статистическую структуру. Выделение этой структуры — как раз то, в чем преуспели алгоритмы машинного обучения. Модели машинного обучения могут изучать скрытое статистическое пространство изображений, музыки и литературных произведений, а затем, основываясь на образцах из этого пространства, создавать новые произведения с характеристиками, схожими с теми, которые модель видела в обучающих данных. Естественно, создание таких произведений трудно назвать актом творчества. Это простая математическая операция: алгоритм не имеет опыта человеческой жизни, человеческих эмоций или нашего практического опыта; он учится на опыте, который имеет мало общего с нашим. Это только наша интерпретация как наблюдателей, придающая смысл тому, что генерирует модель. Но в руках опытного художника алгоритм может стать управляемым инструментом создания наполненных смыслом и прекрасных произведений. Скрытое пространство образцов может стать кистью, наделяющей художника новыми возможностями и расширяющей пространство нашего воображения. Более того, ИИ может сделать художественное творчество более доступным, избавляя от необходимости обладать техническими и практическими навыками — создавая новую среду чистого искусства без присесла.

Янис Ксенакис (Iannis Xenakis), пионер электронной и алгоритмической музыки, прекрасно выразил ту же идею в 1960-х в контексте применения технологий автоматизации к музыкальной композиции<sup>1</sup>:

*Свободный от утомительных вычислений, композитор способен посвятить себя общим проблемам, которые создает новая музыкальная форма, и исследовать самые потаенные уголки этой формы, изменяя значения входных данных. Например, он может испытывать все инструментальные комбинации, от одиночных инструментов до крупных оркестров. С помощью электронных компьютеров композитор может стать чем-то вроде пилота: он нажимает кнопки, вводит координаты и управляет космическим кораблем, плывущим в пространстве звуков, через звуковые созвездия и галактики, которые прежде он мог видеть только во снах.*

В этой главе мы с разных сторон рассмотрим потенциал глубокого обучения для расширения творческих возможностей. Мы рассмотрим приемы генерирования

---

<sup>1</sup> Iannis Xenakis, «Musiques formelles: nouveaux principes formels de composition musicale», специальный выпуск *La Revue musicale*, nos. 253–254 (1963).

последовательностей данных (которые можно использовать для создания текста или музыки), алгоритм DeepDream и методы создания изображений с использованием вариационных автокодировщиков и генеративно-состязательных сетей. Мы заставим ваш компьютер придумывать новые произведения, никогда не виданные прежде; и может быть, вы тоже станете мечтать о фантастических возможностях, лежащих на пересечении технологии и искусства. Приступим.

## 8.1. Генерирование текста с помощью LSTM

В этом разделе мы посмотрим, как можно использовать рекуррентные нейронные сети для генерирования последовательностей данных. В качестве примера мы будем генерировать текст, однако представленные здесь методы можно распространить на любые последовательные данные: вы можете применить их к последовательности музыкальных нот и получить новую музыку или к последовательности данных, описывающих мазки кистью (например, записанных в процессе рисования художником на iPad), и сгенерировать картину мазок за мазком и т. д.

Генерирование последовательностей данных не ограничивается созданием художественных произведений. Этот прием с успехом используется для синтеза речи и генерирования диалогов для чат-ботов. Функция Smart Reply, представленная компанией Google в 2016 году и способная автоматически генерировать короткие ответы на электронные письма или текстовые сообщения, основана на подобных приемах.

### 8.1.1. Краткая история генеративных рекуррентных сетей

В конце 2014 года мало кто был знаком с аббревиатурой LSTM даже в сообществе машинного обучения. Успешное применение методов генерации последовательностей данных с помощью рекуррентных сетей начало приобретать широкую известность только в 2016 году. Но сами методы имеют довольно давнюю историю, начиная с разработки алгоритма LSTM в 1997 году<sup>1</sup>. Этот новый алгоритм первое время использовался для генерации текстов символ за символом.

В 2002 году Дуглас Эк (Douglas Eck), а затем и исследователи в швейцарской лаборатории им. Шмидхубера (Schmidhuber) применили алгоритм LSTM для генерации музыки и получили многообещающие результаты. В настоящее время Эк занимается исследованиями в подразделении Google Brain. В 2016 году он основал новую исследовательскую группу, получившую название Magenta, и сосредоточился на применении современных методов глубокого обучения для создания привлекательной музыки. Иногда хорошей идеи требуется 15 лет, чтобы превратиться в осозаемый результат.

<sup>1</sup> Sepp Hochreiter and Jürgen Schmidhuber, «Long Short-Term Memory», *Neural Computation* 9, no. 8 (1997).

В конце 2000-х – начале 2010-х Алекс Грэйвз (Alex Graves) проделал важную новаторскую работу по использованию рекуррентных сетей для генерации последовательностей данных. В частности, в 2013 году он работал над комбинацией рекуррентной и полносвязной сетей для получения человекоподобного почерка, используя временные последовательности позиций ручки, и эта работа расценивается некоторыми как поворотный момент<sup>1</sup>. Так, это конкретное применение нейронных сетей именно в этот момент времени открыло мне понятие *мечтающих машин* и подтолкнуло начать разработку фреймворка Keras. В 2013 году Грэйвз оставил похожее закомментированное замечание, скрытое в файле LaTeX, выгруженном на сервер препринтов arXiv: «генерация последовательностей данных – это самая близкая к воплощению мечта компьютеров». Несколько лет спустя мы принимаем такие разработки как нечто само собой разумеющееся; однако в то время трудно было наблюдать за демонстрациями Грэйвза и не приходить в восторг от открывавшихся возможностей.

С тех пор рекуррентные сети с успехом используются для генерации музыки, диалогов, изображений, синтеза речи и проектирования молекул. Они даже использовались для создания сценария фильма, роли в котором исполняли живые актеры.

### 8.1.2. Как генерируются последовательности данных?

Универсальный способ генерации последовательностей данных с применением методов глубокого обучения заключается в обучении сети (обычно рекуррентной или сверточной сети) для прогнозирования следующего токена или следующих нескольких токенов в последовательности, опираясь на предыдущие токены. Например, для входной последовательности «the cat is on the ma,» сеть обучается предсказывать целевую букву  $t^2$ , следующий символ. Как обычно, при работе с текстовыми данными в роли *токенов* часто выступают слова или символы, и любая сеть, моделирующая вероятность появления следующего токена на основе предыдущих, называется *языковой моделью*. Языковая модель фиксирует *скрытое пространство языка* – его статистическую структуру.

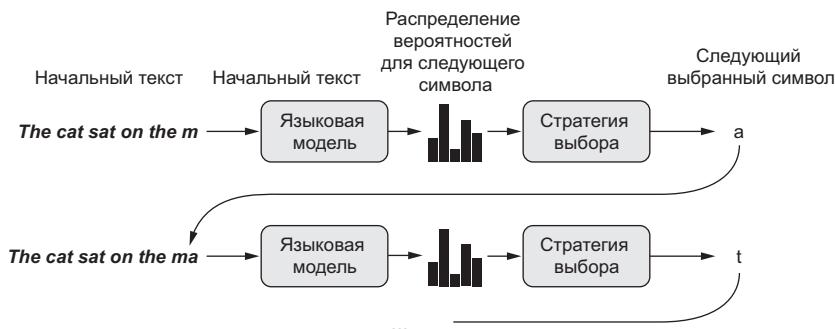
После получения такой обученной языковой модели мы можем *извлекать образцы* из нее (генерировать новые последовательности): передать ей начальную строку текста (так называемые *кондиционные данные*), попросить сгенерировать следующий символ или слово (можно даже сгенерировать несколько слов сразу), добавить сгенерированный вывод в конец предыдущих входных данных и повторить процесс много раз (рис. 8.1). Этот цикл позволяет генерировать последовательности произвольной длины, отражающие структуру данных, на которых обучалась модель: последовательности, которые выглядят *почти* как предложения, написанные чело-

---

<sup>1</sup> Alex Graves, «Generating Sequences With Recurrent Neural Networks», arXiv (2013), <https://arxiv.org/abs/1308.0850>.

<sup>2</sup> В результате получается «the cat is on the mat» – «кошка на коврике». – Примеч. пер.

веком. В примере, представленном далее в этом разделе, мы возьмем слой LSTM, передадим ему строки длиной  $N$  символов, извлеченные из текстового корпуса, и обучим его предсказывать символ  $N + 1$ . На выходе модель будет возвращать вектор softmax с вероятностями для всех возможных символов: распределение вероятностей для следующего символа. Такой слой LSTM называется *языковой нейронной моделью уровня символов*.



**Рис. 8.1.** Процесс посимвольной генерации текста с использованием языковой модели

### 8.1.3. Важность стратегии выбора

Для генерации текста важную роль играет алгоритм выбора следующего символа. Наивное решение — *жадный выбор*, когда выбирается наиболее вероятный символ. Но такой подход приводит к получению в результате повторяющихся, предсказуемых строк, которые не выглядят связанными предложениями. Намного интереснее подход, который делает порой неожиданный выбор, вводя случайную составляющую в процесс выбора из распределения вероятностей для следующего символа. Этот подход называется *стохастическим выбором* (как вы помните, слово *стохастический* в данном контексте является синонимом слова *случайный*). Таким образом, если символ *e* имеет вероятность 0,3 стать следующим символом, согласно модели мы выберем его в 30 % случаев. Обратите внимание на то, что жадный выбор тоже может использоваться для выбора из распределения вероятностей, когда какой-то символ имеет вероятность 1, а все остальные — вероятность 0.

Вероятностный выбор из вектора softmax, возвращаемого моделью, является хорошим решением: он позволяет время от времени появляться в выводе даже маловероятным символам, генерировать более интересные предложения и иногда демонстрировать творческую жилку, придумывая новые, реалистично звучащие слова, которые отсутствуют в обучающих данных. Однако здесь есть одна проблема: эта стратегия не предусматривает возможности *управлять величиной случайности* в процессе выбора.

Зачем может понадобиться увеличивать или уменьшать случайную составляющую? Рассмотрим крайний случай: чисто случайный выбор, когда следующий

символ выбирается из равномерно распределенных вероятностей и каждый символ одинаково вероятен. Эта схема имеет максимальную случайность; иными словами, это распределение вероятностей имеет максимальную энтропию. Естественно, она не произведет ничего интересного. С другой стороны, жадный выбор тоже не производит ничего интересного и не имеет случайной составляющей: соответствующее распределение вероятностей имеет минимальную энтропию. Выбор из «реального» распределения вероятностей — распределения, возвращаемого функцией softmax, — находится между этими двумя крайностями. Но есть еще множество других промежуточных точек с большей или меньшей энтропией, которые вы, возможно, захотите исследовать. Меньшая энтропия позволит генерировать последовательности с более предсказуемой структурой (и поэтому выглядящие более реалистичными), тогда как большая энтропия даст более неожиданный и творческий результат. Выбирая результаты из генеративных моделей, всегда полезно исследовать разные величины случайности в процессе генерации. Поскольку высшими судьями, определяющими, насколько интересны сгенерированные данные, являются мы, люди, интересность оказывается весьма субъективной величиной, и поэтому нельзя сказать наперед, где лежит точка оптимальной энтропии.

Для управления величиной случайности в процессе выбора введем параметр, который назовем *температурой softmax*, характеризующий энтропию распределения вероятностей, используемую для выбора: она будет определять степень необычности или предсказуемости выбора следующего символа. С учетом значения *temperature* и на основе оригинального распределения вероятностей (результата функции softmax модели) будет вычисляться новое распределение путем взвешивания вероятностей, как показано ниже.

### Листинг 8.1. Взвешивание распределения вероятностей с учетом значения температуры

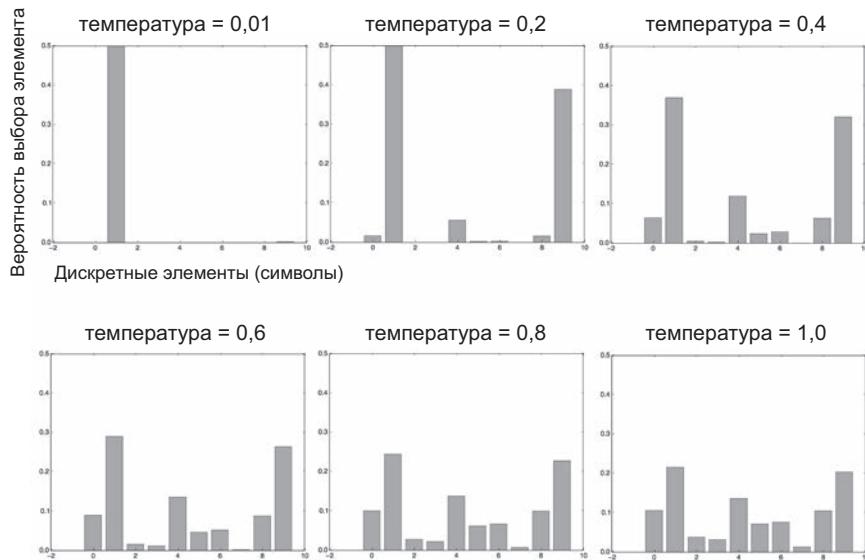
```
import numpy as np

→ def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

original\_distribution — это одномерный массив NumPy значений вероятностей, сумма которых должна быть равна 1

Возвращает новую, взвешенную версию оригинального распределения. Сумма вероятностей в новом распределении может получиться больше 1, поэтому разделим элементы вектора на сумму, чтобы получить новое распределение

Чем выше температура, тем больше энтропия распределения вероятностей и тем более неожиданными и менее структурированными будут генерируемые данные. Чем меньше температура, тем меньше будет величина случайной составляющей и тем более предсказуемыми будут генерируемые данные (рис. 8.2).



**Рис. 8.2.** Разные результаты взвешивания одного и того же распределения вероятностей. Низкая температура = высокая предсказуемость, высокая температура = более случайный результат

#### 8.1.4. Реализация посимвольной генерации текста на основе LSTM

Воплотим эти идеи на практике в реализации с Keras. Первое, что нам понадобится, — это много текстовых данных, на которых можно было бы обучить языковую модель. Для этого можно использовать любой большой текстовый файл или набор текстовых файлов, например статьи из Википедии, роман «Властелин колец» и т. д. В данном примере мы используем тексты из произведений Ницше, немецкого философа конца XIX века (в переводе на английский язык). Таким образом, в результате обучения у нас получится языковая модель, обладающая специфическими особенностями, характерными для произведений Ницше, а не обобщенная модель английского языка.

##### Подготовка данных

Для начала загрузим корпус и преобразуем текст в нижний регистр.

##### Листинг 8.2. Загрузка и парсинг исходного текстового файла

```
import keras
import numpy as np

path = keras.utils.get_file(
```

Продолжение ↗

**Листинг 8.2** (продолжение)

```
'nietzsche.txt',
origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = open(path).read().lower()
print('Corpus length:', len(text))
```

Затем извлечем частично перекрывающиеся последовательности с длиной maxlen, выполним прямое кодирование и упакуем в трехмерный массив NumPy x с формой (последовательности, максимальная\_длина, уникальные\_символы). Одновременно подготовим массив y с соответствующими целями: векторы с символами, полученные прямым кодированием, которые следуют за каждой извлеченной последовательностью.

**Листинг 8.3.** Векторизация последовательностей символов

```
maxlen = 60 ← Извлечение последовательностей по 60 символов
step = 3 ← Новые последовательности выбираются через каждые 3 символа
sentences = [] ← Хранение извлеченных последовательностей
next_chars = [] ← Хранение целей (символов, следующих за последовательностями)

for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])

print('Number of sequences:', len(sentences))

chars = sorted(list(set(text))) ← Список уникальных символов в корпусе
print('Unique characters:', len(chars))
char_indices = dict((char, chars.index(char)) for char in chars) ←

print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1 ←
```

Прямое кодирование символов  
в бинарные массивы

Словарь, отображающий  
уникальные символы в их  
индексы в списке «chars»

## Конструирование сети

Эта сеть состоит из единственного слоя LSTM, за которым следует классификатор Dense с функцией softmax выбора из всех возможных символов. Но имейте в виду, что рекуррентные нейронные сети не единственный способ генерирования последовательностей данных; одномерные сверточные сети тоже показали превосходные результаты в решении этой задачи.

**Листинг 8.4.** Модель с единственным слоем LSTM для предсказания следующего символа

```
from keras import layers

model = keras.models.Sequential()
model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))
```

Так как цели имеют формат прямого кодирования, используем для обучения модели функцию потерь `categorical_crossentropy`.

**Листинг 8.5.** Конфигурация компилируемой модели

```
optimizer = keras.optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

**Обучение модели и извлечение образцов из нее**

Имея обученную модель и фрагмент начального текста, можно сгенерировать новый текст, выполнив следующие пункты:

1. Извлечь из модели распределение вероятностей следующего символа для имеющегося на данный момент сгенерированного текста.
2. Выполнить взвешивание распределения с заданной температурой.
3. Выбрать следующий символ в соответствии с вновь взвешенным распределением вероятностей.
4. Добавить новый символ в конец текста.

Вот код, который мы используем для взвешивания оригинального распределения вероятностей, возвращаемого моделью, и извлечения индекса символа (*функция выборки*).

**Листинг 8.6.** Функция выборки следующего символа с учетом прогнозов модели

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Наконец, следующий цикл повторяет обучение и генерирует текст. Для начала генерируем текст, использовав разные температуры после каждой эпохи. Это позволит вам увидеть, как меняется генерируемый тест по мере схождения модели и как температура влияет на стратегию выбора.

**Листинг 8.7.** Цикл генерации текста

```

import random           ← Обучение
import sys              ← модели
for epoch in range(1, 60): ← в течение
    print('epoch', epoch) ← 60 эпох
    model.fit(x, y, batch_size=128, epochs=1) ← Выполнение одной
                                                итерации обучения

start_index = random.randint(0, len(text) - maxlen)
generated_text = text[start_index: start_index + maxlen]
print('--- Generating with seed: "' + generated_text + '"') ← Выбор
                                                                случайного
                                                                начального
                                                                текста

for temperature in [0.2, 0.5, 1.0, 1.2]: ← Генерация текста
    print('----- temperature:', temperature) ← для разных температур
    sys.stdout.write(generated_text)

for i in range(400): ← Генерация 400 символов, начиная с начального текста
    sampled = np.zeros((1, maxlen, len(chars))) ← Прямое кодирование
    for t, char in enumerate(generated_text): ← символов, сгенерирован-
        sampled[0, t, char_indices[char]] = 1. ← ных до сих пор

    preds = model.predict(sampled, verbose=0)[0] ← Выбор
    next_index = sample(preds, temperature) ← следующего
    next_char = chars[next_index] ← символа

    generated_text += next_char
    generated_text = generated_text[1:]

    sys.stdout.write(next_char)

```

Мы выбрали случайный начальный текст «new faculty, and the jubilation reached its climax when kant»<sup>1</sup>. Далее показано, что получилось после 20-й эпохи, задолго до того, как модель полностью сошлась, для `temperature=0.2`:

```

new faculty, and the jubilation reached its climax when kant and such a man
in the same time the spirit of the surely and the such the such
as a man is the sunligh and subject the present to the superiority of the
special pain the most man and strange the subjection of the
special conscience the special and nature and such men the subjection of the
special men, the most surely the subjection of the special
intellect of the subjection of the same things and

```

Вот результат для `temperature=0.5`:

```

new faculty, and the jubilation reached its climax when kant in the eterned
and such man as it's also become himself the condition of the
experience of off the basis the superiory and the special morty of the
strength, in the langus, as which the same time life and "even who
discless the mankind, with a subject and fact all you have to be the stand

```

<sup>1</sup> «...новая способность сделалась даже причиной чрезвычайного возбуждения, и ликование достигло своего апогея, когда Кант...» (Фридрих Ницше, «По ту сторону добра и зла»). — Примеч. пер.

and lave no comes a trovation of the man and surely the  
conscience the superiority, and when one must be w

А это результат для temperature=1.0:

new faculty, and the jubilation reached its climax when kant, as a  
periliting of manner to all definites and transpects it it so  
hicable and ont him artiar result  
too such as if ever the proping to makes as cneience. to been juden,  
all every could coldiciousnike hother aw passife, the plies like  
which might thiod was account, indifferent germin, that everythery  
certain destruction, intellect into the deteriorablen origin of moralian,  
and a lessority o

После 60 эпох модель полностью сошлась и текст начал выглядеть более согласованным. Вот результат для temperature=0.2:

cheerfulness, friendliness and kindness of a heart are the sense of the  
spirit is a man with the sense of the sense of the world of the  
self-end and self-concerning the subjection of the strengthorixes--the  
subjection of the subjection of the subjection of the  
self-concerning the feelings in the superiority in the subjection of the  
subjection of the spirit isn't to be a man of the sense of the  
subjection and said to the strength of the sense of the

Для temperature=0.5:

cheerfulness, friendliness and kindness of a heart are the part of the soul  
who have been the art of the philosophers, and which the one  
won't say, which is it the higher the and with religion of the frences.  
the life of the spirit among the most continuess of the  
strengthorixes of the sense the conscience of men of precisely before enough  
presumption, and can mankind, and something the conceptions, the  
subjection of the sense and suffering and the

И для temperature=1.0:

cheerfulness, friendliness and kindness of a heart are spiritual by the  
ciutre for the  
entalled is, he astraged, or errors to our you idstood--and it needs,  
to think by spars to whole the amvives of the newoatly, prefectly  
raals! it was  
name, for example but voludd atu-especity"--or rank onee, or even all  
"solett increessic of the world and  
implussional tragedy experience, transf, or insiderar,--must hast  
if desires of the strubction is be stronges

Как видите, при низком значении температуры генерируется часто повторяющийся и предсказуемый текст, однако локальная структура очень реалистична: в частности, все слова (слово является локальным шаблоном символов) — это

действительные английские слова. При высоком значении температуры генерируется более интересный текст, неожиданный и даже творческий; в нем иногда появляются совершенно новые слова, кажущиеся правдоподобными (например, *etermed* и *troveration*). При высокой температуре внутренняя структура начинает разрушаться, и большинство слов выглядят как полуслучайные последовательности символов. Без сомнения, 0,5 — самая интересная температура для генерации текста в данном конкретном решении. Всегда пробуйте несколько стратегий выбора! Разумный баланс между изученной структурой и случайностью — вот что делает сгенерированные данные интересными.

Обратите внимание на то, что, обучая модель большего размера дольше и на большем объеме данных, можно добиться генерации текста, который выглядит еще реалистичнее. Однако не думайте, что когда-нибудь вам удастся сгенерировать осмысленный текст, разве только по чистой случайности: вы всего лишь выбираете образцы данных из статистической модели, в которой символы следуют за символами. Язык — это канал общения, и существуют различия между сутью общения и статистической структурой сообщений, которыми кодируется общение. Чтобы осознать это различие, проведите мысленный эксперимент: представьте, что человеческий язык позволял бы сжимать информацию при общении почти так же, как это делают компьютеры с цифровой информацией. Язык не потерял бы своей осмысленности, но утратил статистическую структуру, что сделало бы невозможным обучение языковой модели, как мы только что это проделали.

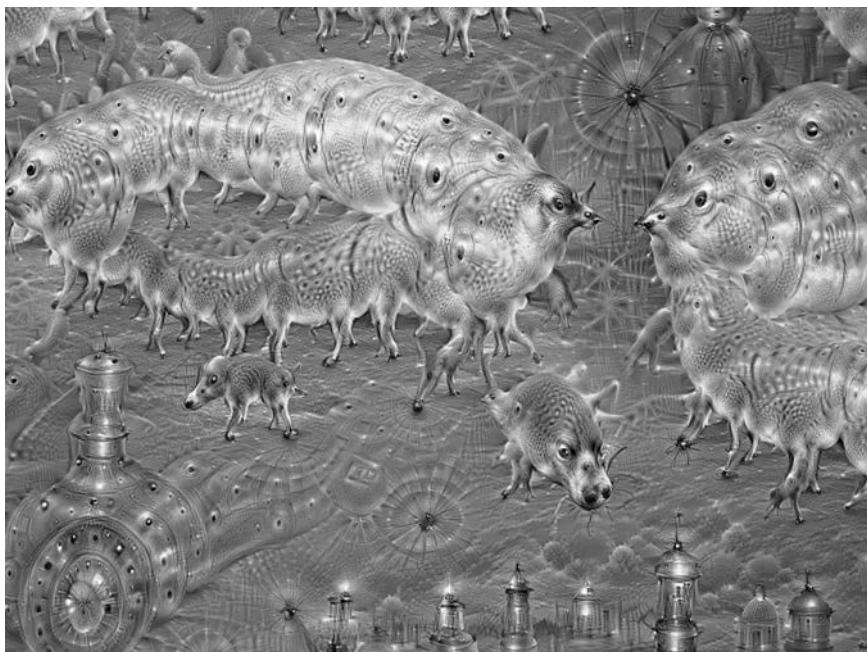
### 8.1.5. Подведение итогов

- ❑ Обучая модель для предсказания следующего токена по предшествующим, можно генерировать последовательности дискретных данных.
- ❑ В случае с текстом такая модель называется *языковой моделью*; она может быть основана на словах или символах.
- ❑ Выбор следующего токена требует баланса между мнением модели и случайностью.
- ❑ Обеспечить такой баланс можно введением понятия температуры; всегда пробуйте разные температуры, чтобы найти правильную.

## 8.2. DeepDream

*DeepDream* — это метод художественной обработки изображений, основанный на использовании представлений, полученных сверточными нейронными сетями. Впервые он был реализован в компании Google летом 2015 года как демонстрация возможностей библиотеки глубокого обучения Caffe (она появилась за несколько

месяцев до выхода первой общедоступной версии TensorFlow)<sup>1</sup>. В интернете он быстро превратился в сенсацию благодаря получаемым с его помощью психodelическим картинам (см., например, рис. 8.3), наполненным алгоритмическими иллюзиями, птичьими перьями и собачьими глазами — побочный эффект обучения сверточной сети DeepDream на изображениях из ImageNet, где породы собак и виды птиц представлены шире всего.



**Рис. 8.3.** Пример изображения, созданного с помощью DeepDream

Алгоритм DeepDream почти идентичен методу визуализации фильтров сверточных сетей, представленному в главе 5, и состоит из сверточной сети, действующей в обратном направлении: выполняет градиентное восхождение по входным данным, максимизируя активацию определенного фильтра на более высоком слое сверточной сети. DeepDream использует ту же идею с небольшими различиями:

- ❑ алгоритм DeepDream пытается максимизировать активацию всех слоев, а не только определенного фильтра, тем самым смешивая визуализации большего количества признаков;

<sup>1</sup> Alexander Mordvintsev, Christopher Olah, and Mike Tyka, «DeepDream: A Code Example for Visualizing Neural Networks», *Google Research Blog*, July 1, 2015, <http://mng.bz/xXLM>.

- вы начинаете не на пустом месте, со случайных входных данных, а с имеющегося изображения, в результате получающиеся эффекты замыкаются на существующие визуальные шаблоны, искажая элементы изображения на художественный манер;
- входные изображения обрабатываются в разных масштабах (называемых *октавами*), что улучшает качество визуализации.

Давайте немного поэкспериментируем с DeepDream.

### 8.2.1. Реализация DeepDream в Keras

Начнем со сверточной сети, предварительно обученной на наборе ImageNet. В Keras имеется несколько таких сетей: VGG16, VGG19, Xception, ResNet50 и т. д. Реализовать DeepDream можно с любой из них, но, как вы понимаете, выбор повлияет на характер визуализаций, потому что разные сверточные архитектуры выделяют из исходных данных разные признаки. В оригинальной версии DeepDream использовалась модель Inception, и на практике эта модель известна красиво выглядящими картинками DeepDreams, поэтому мы используем модель Inception V3, входящую в состав Keras.

#### Листинг 8.8. Загрузка предварительно обученной модели Inception V3

```
from keras.applications import inception_v3
from keras import backend as K
K.set_learning_phase(0)                                ← Мы не будем обучать модель,
                                                       поэтому выполним данную команду,
                                                       чтобы запретить все операции,
                                                       имеющие отношение к обучению
model = inception_v3.InceptionV3(weights='imagenet',   | Конструирование сети
                                   include_top=False)      | Inception V3 без сверточной
                                                       | основы. Модель будет загружаться
                                                       | с весами, полученными
                                                       | в результате предварительного
                                                       | обучения на наборе ImageNet
```

Теперь вычислим *потери*: величину, которую мы будем максимизировать в процессе градиентного восхождения. В главе 5, обсуждая визуализацию фильтров, мы пытались максимизировать значение определенного фильтра в определенном слое. Здесь мы будем максимизировать одновременно активации всех фильтров в нескольких слоях. В данном случае максимизироваться будет взвешенная сумма L2-норм активаций набора верхних слоев. Точный набор выбранных слоев (а также их вклад в окончательное значение потерь) оказывает большое влияние на производимые визуальные эффекты, поэтому мы должны сделать эти параметры легко настраиваемыми. Нижние слои порождают геометрические шаблоны, а верхние создают эффекты, в которых можно распознать некоторые классы из набора ImageNet (например, птицы или собаки). Начнем с произвольно выбранной конфигурации, состоящей из четырех слоев, но позднее вы наверняка захотите исследовать другие конфигурации.

**Листинг 8.9.** Настройка конфигурации DeepDream

```
layer_contributions = {
    'mixed2': 0.2,
    'mixed3': 3.,
    'mixed4': 2.,
    'mixed5': 1.5,
}
```

Словарь отображает имена слоев в коэффициенты, определяющие вклады слоев в потери, которые мы будем максимизировать. Обратите внимание: имена слоев жестко «зашиты» во встроенное приложение Inception V3. Получить список имен всех слоев можно с помощью `model.summary()`

Теперь определим тензор, содержащий потери: взвешенную сумму L2-норм активаций слоев из листинга 8.9.

**Листинг 8.10.** Определение потерь для максимизации

```
layer_dict = dict([(layer.name, layer) for layer in model.layers])
```

Создается словарь, отображающий имена слоев в их экземпляры

Величина потерь определяется добавлением вклада слоя в эту скалярную переменную

```
loss = K.variable(0.)
for layer_name in layer_contributions:
    coeff = layer_contributions[layer_name]
    activation = layer_dict[layer_name].output ← Получение результата слоя
```

scaling = K.prod(K.cast(K.shape(activation), 'float32'))

Добавление L2 нормы признаков слоя к потерям. Чтобы избежать влияния рамок, в подсчете потерь участвуют только пиксели, не попадающие на рамку

```
→ loss += coeff * K.sum(K.square(activation[:, 2: -2, 2: -2, :])) / scaling
```

Теперь можно настроить процесс градиентного восхождения.

**Листинг 8.11.** Процесс градиентного восхождения

```
dream = model.input ← Этот тензор хранит сгенерированное изображение
```

grads = K.gradients(loss, dream)[0] ← Вычисление градиентов изображения с учетом потерь

grads /= K.maximum(K.mean(K.abs(grads)), 1e-7) ← Нормализация градиентов (важный шаг)

outputs = [loss, grads]
fetch\_loss\_and\_grads = K.function([dream], outputs)

```
def eval_loss_and_grads(x):
    outs = fetch_loss_and_grads([x])
    loss_value = outs[0]
    grad_values = outs[1]
    return loss_value, grad_values
```

Настройка функции Keras для извлечения значения потерь и градиентов для заданного исходного изображения

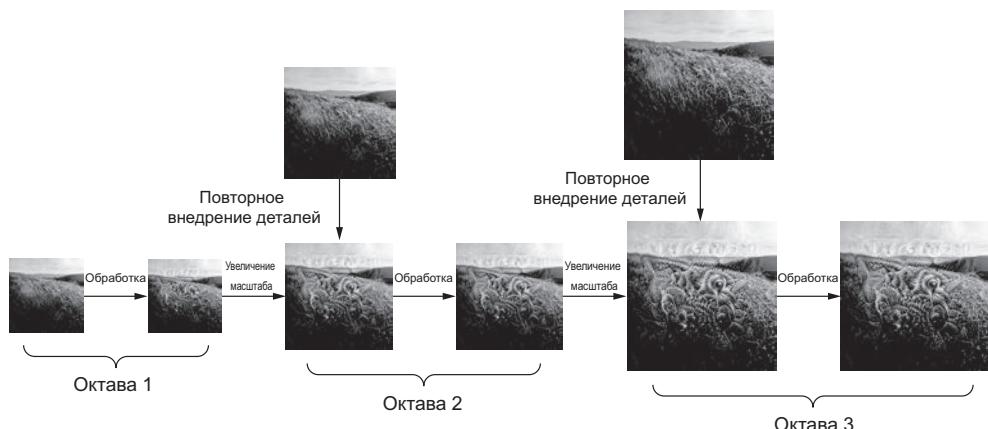
Продолжение ↗

## Листинг 8.11 (продолжение)

```
def gradient_ascent(x, iterations, step, max_loss=None):
    for i in range(iterations):
        loss_value, grad_values = eval_loss_and_grads(x)
        if max_loss is not None and loss_value > max_loss:
            break
        print('...Loss value at', i, ':', loss_value)
        x += step * grad_values
    return x
```

Эта функция выполняет заданное число итераций градиентного восхождения

Наконец, фактический алгоритм DeepDream. Сначала определим список масштабов (также называемых октавами), в которых будут обрабатываться изображения. Каждый следующий больше предыдущего в 1,4 раза (на 40 % крупнее): начнем с обработки маленького изображения и затем постепенно будем увеличивать его (рис. 8.4).



**Рис. 8.4.** Процесс DeepDream: последовательные масштабы пространственной обработки (октавы) и повторное внедрение деталей при увеличении масштаба

Для каждого последующего масштаба, от меньшего к большему, выполняется градиентное восхождение для максимизации функции потерь, определенной прежде. После каждого цикла восхождения полученное изображение увеличивается на 40 %.

Чтобы избежать потери деталей изображения после каждого увеличения масштаба (в результате чего появляются эффекты размытия или мозаичности), можно использовать простой прием: после каждого изменения масштаба повторно внедрять в изображение потерянные детали, что возможно благодаря знанию, как должно выглядеть исходное изображение в увеличенном масштабе. Имея маленькое изображение с размером  $S$  и большое — с размером  $L$ , можно вычислить разницу между оригинальным изображением с увеличенным размером  $L$  и оригинальным изображением с уменьшенным размером  $S$  — эта разница количественно отражает потерянные детали при переходе от размера  $S$  к размеру  $L$ .

**Листинг 8.12.** Выполнение градиентного восхождения через последовательность масштабов

```

Изменяя гиперпараметры можно добиваться новых эффектов
import numpy as np

step = 0.01
num_octave = 3
octave_scale = 1.4
iterations = 20
max_loss = 10.
base_image_path = '...'

# Изменяя гиперпараметры можно добиваться новых эффектов
# Размер шага градиентного восхождения
# Количество масштабов, на которых выполняется градиентное восхождение
# Отношение между соседними масштабами
# Число шагов восхождения для каждого масштаба
# Если величина потерь вырастет больше 10, мы должны прервать процесс градиентного восхождения, чтобы избежать появления безобразных артефактов
# Укажите здесь путь к файлу изображения

img = preprocess_image(base_image_path) # Загрузка базового изображения в массив NumPy (функция определена в листинге 8.13)

original_shape = img.shape[1:3]
successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)
# Подготовка списка кортежей shape, определяющих разные масштабы для выполнения градиентного восхождения

successive_shapes = successive_shapes[::-1] # Переворачивание списка кортежей shape, чтобы они следовали в порядке возрастания масштаба

original_img = np.copy(img)
shrunk_original_img = resize_img(img, successive_shapes[0]) # Уменьшение изображения в массиве NumPy до наименьшего масштаба

for shape in successive_shapes:
    print('Processing image shape', shape)
    img = resize_img(img, shape) # Увеличение масштаба изображения
    img = gradient_ascent(img,
                          iterations=iterations,
                          step=step,
                          max_loss=max_loss) # Выполнение градиентного восхождения с изменением изображения

    upscaled_shrunk_original_img = resize_img(shrunk_original_img, shape)
    same_size_original = resize_img(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img # Разница между двумя изображениями — это детали, утерянные в результате масштабирования

    img += lost_detail # Повторное внедрение деталей
    shrunk_original_img = resize_img(original_img, shape)
    save_img(img, fname='dream_at_scale_' + str(shape) + '.png') # Вычисление высококачественной версии исходного изображения с заданными размерами

    save_img(img, fname='final_dream.png')

```

Код в листинге 8.12 использует следующие вспомогательные функции с говорящими именами, которые манипулируют массивами NumPy. Они требуют установки пакета SciPy.

### Листинг 8.13. Вспомогательные функции

```
import scipy
from keras.preprocessing import image

def resize_img(img, size):
    img = np.copy(img)
    factors = (1,
               float(size[0]) / img.shape[1],
               float(size[1]) / img.shape[2],
               1)
    return scipy.ndimage.zoom(img, factors, order=1)

def save_img(img, fname):
    pil_img = deprocess_image(np.copy(img))
    scipy.misc.imsave(fname, pil_img)

def preprocess_image(image_path):
    img = image.load_img(image_path)
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = inception_v3.preprocess_input(img)
    return img

def deprocess_image(x):
    if K.image_data_format() == 'channels_first':
        x = x.reshape((3, x.shape[2], x.shape[3]))
        x = x.transpose((1, 2, 0))
    else:
        x = x.reshape((x.shape[1], x.shape[2], 3))
        x /= 2.
        x += 0.5
        x *= 255.
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Вспомогательная функция,  
открывающая изображение,  
изменяющая его размер  
и преобразующая изображение  
в тензор для обработки в Inception V3

Вспомогательная функция,  
преобразующая тензор  
в допустимое изображение

Отмена предварительной  
обработки, выполненной  
вызовом `inception_`  
`v3.preprocess_input`

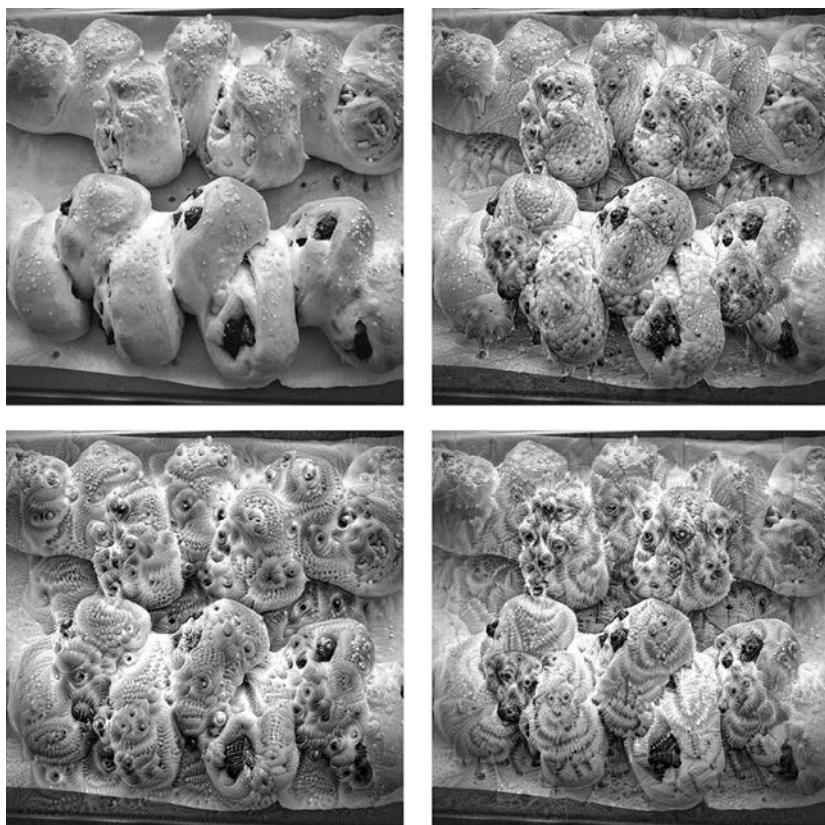
### ПРИМЕЧАНИЕ

Оригинальная сеть Inception V3 была обучена распознавать образы на изображениях размером  $299 \times 299$ , поэтому, с учетом того, что процесс выполняет уменьшение масштаба изображений с разумным коэффициентом, реализация DeepDream производит лучшие результаты для изображений с размерами между  $300 \times 300$  и  $400 \times 400$ . Тем не менее вы можете использовать тот же код для обработки изображений любого размера, с любым соотношением сторон.

Начав с фотографии, сделанной на невысоких холмах между заливом Сан-Франциско и кампусом Google, мы получили результат, изображенный на рис. 8.5.



**Рис. 8.5.** Результат обработки изображения реализацией алгоритма DeepDream



**Рис. 8.6.** Применение разных конфигураций DeepDream к изображению

Мы настоятельно рекомендуем вам попробовать разные настройки слоев, которые используются для определения потерь. Слои, находящиеся в сети ниже, содержат более локальные, менее абстрактные представления и порождают более геометрические шаблоны. Слои, находящиеся выше, порождают эффекты, в которых можно распознать объекты, наиболее часто встречающиеся в наборе ImageNet, такие как собачьи глаза, перья птиц и т. д. Вы можете организовать случайный перебор параметров в словаре `layer_contributions`, чтобы быстро оценить множество разных комбинаций слоев. На рис. 8.6 показан диапазон результатов, полученных с использованием разных конфигураций слоев из изображения с домашними булочками.

### 8.2.2. Подведение итогов

- Алгоритм DeepDream состоит из сверточной сети, действующей в обратном направлении и генерирующей входные данные на основе представлений, полученных в результате обучения.
- Получаемые результаты выглядят забавно и напоминают визуальные галлюцинации, возникающие у людей, страдающих нарушением работы зрительного отдела коры головного мозга.
- Обратите внимание на то, что этот процесс не является специфическим для моделирования изображений или даже для сверточных сетей. Его можно применить к речи, музыке и т. д.

## 8.3. Нейронная передача стиля

Кроме DeepDream, существует еще одна важная разработка в области изменения изображений с использованием глубокого обучения — *нейронная передача стиля*, реализованная Леоном Гатисом (Leon Gatys) с коллегами летом 2015 года<sup>1</sup>. После своего появления алгоритм нейронной передачи стиля претерпел множество усовершенствований, породил множество вариаций и нашел применение во множестве приложений обработки фотографий для смартфонов. Для простоты в этом разделе основное внимание уделяется формулировке из оригинальной статьи.

Нейронная передача стиля заключается в применении стиля изображения-образца к целевому изображению при сохранении содержимого этого целевого изображения. Пример передачи стиля изображен на рис. 8.7.

В данном контексте под *стилем* в основном подразумеваются текстуры, цветовая палитра и визуальные шаблоны в различных пространственных масштабах, а под *содержимым* — высокоуровневая макроструктура изображения. Например, сине-желтые круговые мазки на рис. 8.7 соответствуют стилю (в качестве образца ис-

<sup>1</sup> Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, «A Neural Algorithm of Artistic Style», arXiv (2015), <https://arxiv.org/abs/1508.06576>.

пользована картина Винсента Ван Гога «Звездная ночь»), а здания на фотографии, сделанной фотографом Тюбингеном (Tübingen), — это содержимое.



**Рис. 8.7.** Пример передачи стиля

Идея передачи стиля, тесно связанная с созданием текстур, давно вынашивалась в сообществе людей, увлеченных обработкой изображений, прежде чем воплотилась в алгоритм нейронной передачи стиля в 2015 году. Однако, как оказалось, реализации передачи стиля, основанные на глубоком обучении, не имеют аналогов среди прежних достижений, использовавших классические методики компьютерного зрения, и поэтому они породили удивительный бум в сфере художественных приложений компьютерного зрения.

В основе реализации передачи стиля лежит та же идея, которая занимает центральное положение во всех алгоритмах глубокого обучения: вы задаете функцию потерь, чтобы определить цель для достижения, и минимизируете ее. Вы знаете, чего хотите: сохранить содержимое исходного изображения и передать стиль изображения-образца. Определив математически *содержимое и стиль*, соответствующую функцию потерь для минимизации можно обозначить так:

```
loss = distance(style(reference_image) - style(generated_image)) +
      distance(content(original_image) - content(generated_image))
```

Здесь *distance* — это функция нормы, такой как L2-норма, *content* — функция, принимающая изображение и вычисляющая представление его содержимого, а *style* — функция, принимающая изображение и вычисляющая представление его стиля. Минимизация этой функции потерь приводит к тому, что *style(generated\_image)* приближается к *style(reference\_image)*, а *content(generated\_image)* — к *content(original\_image)*, то есть достигается передача стиля, как мы ее определили.

Фундаментальное наблюдение, сделанное Гатисом с коллегами, заключается в том, что глубокие сверточные нейронные сети дают возможность математически определить функции *style* и *content*. Посмотрим, как это происходит.

### 8.3.1. Функция потерь содержимого

Как вы уже знаете, активации из нижних слоев в сети содержат *локальную* информацию об изображении, тогда как активации из верхних слоев содержат все более

глобальную, абстрактную информацию. Другими словами, активации разных слоев сверточной сети представляют собой разложение содержимого изображения в разных пространственных масштабах. Поэтому можно ожидать, что содержимое более глобального и абстрактного изображения будет захватываться представлениями верхних слоев сети.

Соответственно, хорошим кандидатом на функцию потерь содержимого является L2-норма между активациями верхнего слоя в предварительно обученной сверточной сети, вычисленными по целевому изображению, и активациями того же слоя, вычисленными по сгенерированному изображению. Это гарантирует, как видно из верхнего слоя, что сгенерированное изображение будет выглядеть подобно оригинальному целевому изображению. Если допустить, что верхние слои сверточной сети действительно видят содержимое входных изображений, тогда минимизация этой функции может рассматриваться как способ сохранения содержимого изображения.

### 8.3.2. Функция потерь стиля

Функция потерь содержимого использует только один верхний слой, но функция потерь стиля, согласно определению Гатиса и его коллег, использует несколько слоев сверточной сети: ее цель — захватить внешний вид стиля изображения-образца не в одном, а во всех пространственных масштабах, выделяемых сверточной сетью. В качестве функции потерь стиля Гатис с коллегами используют *матрицу Грама* активаций слоя: внутреннее произведение карт признаков данного слоя. Это внутреннее произведение можно интерпретировать как матрицу корреляций между признаками слоя. Корреляции фиксируют статистики шаблонов определенного пространственного масштаба, которые эмпирически соответствуют текстурам, обнаруженным в этом масштабе.

Следовательно, минимизация функции потерь стиля направлена на сохранение сходных внутренних корреляций между активациями разных слоев изображения-образца и генерируемого изображения. Это, в свою очередь, гарантирует, что текстуры, найденные в разных пространственных масштабах, будут выглядеть одинаково в изображении-образце и сгенерированном изображении.

Проще говоря, предварительно обученную сверточную сеть можно использовать для определения потерь и она будет

- сохранять содержимое, поддерживая сходство активаций верхнего слоя между содержимым целевого и сгенерированного изображений. Сверточная сеть должна «видеть» оба изображения — целевое и сгенерированное — как содержащие одно и то же;
- сохранять стиль, поддерживая сходство корреляций в активациях всех, нижних и верхних, слоев. Корреляции признаков захватывают *текстуры*: изображе-

ние-образец и сгенерированное изображение должны обладать одинаковыми текстурами в разных пространственных масштабах.

Теперь рассмотрим реализацию оригинального алгоритма нейронной передачи стиля 2015 года с применением Keras. Как вы увидите далее, он имеет много общего с реализацией DeepDream, представленной в предыдущем разделе.

### 8.3.3. Нейронная передача стиля в Keras

Нейронную передачу стиля можно реализовать с использованием любой обученной сверточной сети. Здесь мы используем сеть VGG19, которую использовали Гатис с коллегами. VGG19 — это упрощенный вариант сети VGG16, представленной в главе 5, с тремя сверточными слоями.

Вот как выглядит весь процесс в общих чертах:

- Настройка сети, которая вычисляет активации слоя VGG19 одновременно для изображения-образца, целевого и сгенерированного изображений.
- Активации, вычисленные по всем трем изображениям, используются для определения общей функции потерь, описанной выше, которая будет минимизироваться для достижения эффекта передачи стиля.
- Настройка процедуры градиентного восхождения для минимизации этой функции потерь.

Сначала определим пути к изображению-образцу и целевому изображению. Чтобы гарантировать совместимость размеров обрабатываемых изображений (сильно различающиеся размеры затрудняют передачу стиля), приведем их к общей высоте в 400 пикселов.

#### Листинг 8.14. Определение начальных переменных

```
from keras.preprocessing.image import load_img, img_to_array
target_image_path = 'img/portrait.jpg'
style_reference_image_path = 'img/transfer_style_reference.jpg' ← Путь к изображению, которое
width, height = load_img(target_image_path).size
img_height = 400
img_width = int(width * img_height / height) | Размеры
                                                               | генерируемого
                                                               | изображения
                                                               | Путь к изображению с образцом стиля
```

Нам понадобится несколько вспомогательных функций для загрузки, а также для предварительной и заключительной обработки изображений перед передачей изображений в сеть VGG19 и после вывода их из сети.



Теперь определим функцию потерь содержимого, которая позволит гарантировать сходство представлений целевого и сгенерированного изображений в верхнем слое сети VGG19.

#### **Листинг 8.17.** Функция потерь содержимого

```
def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

Далее приводится функция потерь стиля. Она использует вспомогательную функцию для вычисления матрицы Грама из входной матрицы: матрицы корреляций, найденных в матрице оригинальных признаков.

#### **Листинг 8.18.** Функция потерь стиля

```
def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

К этим двум компонентам потерь добавляется третий: функция *общей потери вариации* (total variation loss), которая оперирует пикселями генерируемого изображения. Она стимулирует пространственную целостность генерируемого изображения, что позволяет избежать появления мозаичного эффекта. Ее можно интерпретировать как регуляризацию потерь.

#### **Листинг 8.19.** Функция общей потери вариации

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] -
        x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] -
        x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

Функция потерь, которую мы должны минимизировать, возвращает среднее взвешенное этих трех компонентов. Для вычисления потери содержимого используется только один верхний слой `block5_conv2`, а для вычисления потери содержимого используется список слоев, включающий в себя нижние и верхние слои. Общая потеря вариации добавляется в конец.

В зависимости от используемых изображений с целевым содержимым и образцом стиля, может появиться желание настроить коэффициент `content_weight` (определяет вклад потерь содержимого в общую величину потерь). Большее значение `content_weight` обеспечит большее сходство сгенерированного изображения с целевым.

### Листинг 8.20. Функция общей потери вариации

```

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers]) ← Словарь, отображающий имена
content_layer = 'block5_conv2' ← Слой, используемый для вычисления потерь содержимого
style_layers = ['block1_conv1', ← Слой, используемый для вычисления потерь стиля
               'block2_conv1',
               'block3_conv1',
               'block4_conv1',
               'block5_conv1']
total_variation_weight = 1e-4 ← Веса для вычисления среднего взвешенного по компонентам потерь
style_weight = 1. ← Величина потерь определяется сложением всех компонентов с этой переменной
content_weight = 0.025

loss = K.variable(0.) ← Добавление потери содержимого
layer_features = outputs_dict[content_layer]
target_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss += content_weight * content_loss(target_image_features,
                                         combination_features)

for layer_name in style_layers: ← Добавление потери стиля для каждого целевого уровня
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    loss += (style_weight / len(style_layers)) * sl

loss += total_variation_weight * total_variation_loss(combination_image) ← Добавление общей потери вариации

```

Наконец, настроим процесс градиентного восхождения. В оригинальной статье Гатиса оптимизация выполняется с использованием алгоритма L-BFGS, поэтому мы тоже используем его здесь. Это ключевое отличие от примера DeepDream в разделе 8.2. Реализация алгоритма L-BFGS уже включена в пакет SciPy, однако она имеет два незначительных ограничения:

- ❑ требует передачи значений функции потерь и градиентов в виде двух отдельных функций;

- может применяться только к плоским векторам, тогда как у нас используется трехмерный массив с изображением.

Было бы неэффективно вычислять значения потерь и градиентов независимо, потому что это повлечет большой объем избыточных вычислений; процесс вычисления замедлится почти вдвое по сравнению со случаем, когда эти величины вычисляются вместе. Чтобы обойти эту проблему, определим класс `Evaluator`, вычисляющий значения потерь и градиентов одновременно, который будет возвращать значение потерь при первом обращении и кэшировать градиенты для повторного вызова.

### Листинг 8.21. Подготовка процедуры градиентного спуска

```

Получение градиентов сгенерированного
изображения относительно потерь

grads = K.gradients(loss, combination_image)[0] ←

fetch_loss_and_grads = K.function([combination_image], [loss, grads]) ←

Функция для
получения
значений тек-
ущих потерь
и градиентов

class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height, img_width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grads_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grads_values)
        self.loss_value = None
        self.grads_values = None
        return grad_values

evaluator = Evaluator()

Этот класс обертывает
fetch_loss_and_grads
и позволяет получать
потери и градиенты
вызовами двух от-
дельных методов, как
того требует realiza-
ция оптимизатора
из SciPy

```

Теперь можно запустить процесс градиентного восхождения с использованием реализации алгоритма L-BFGS в SciPy, сохраняя текущее сгенерированное изображение после каждой итерации алгоритма (в данном случае одной итерации соответствуют 20 шагов градиентного восхождения).

**Листинг 8.22.** Цикл передачи стиля

```

from scipy.optimize import fmin_l_bfgs_b
from scipy.misc import imsave
import time

result_prefix = 'my_result'
iterations = 20

x = preprocess_image(target_image_path)
x = x.flatten()
for i in range(iterations):
    print('Start of iteration', i)
    start_time = time.time()
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss,
                                        x,
                                        fprime=evaluator.grads,
                                        maxfun=20)
    print('Current loss value:', min_val)
    img = x.copy().reshape((img_height, img_width, 3))
    img = deprocess_image(img)
    fname = result_prefix + '_at_iteration_%d.png' % i
    imsave(fname, img)
    print('Image saved as', fname)
    end_time = time.time()
    print('Iteration %d completed in %ds' % (i, end_time - start_time))

```

Первичное состояние:  
целевое изображение

Преобразуйте изображение,  
потому что `scipy.optimize.  
fmin_l_bfgs_b` могут  
обрабатывать только плоские  
векторы

Сохраняет текущее  
сгенерированное  
изображение

Выполняет оптимизацию L-BFGS по пикселям генерируемого изображения, чтобы минимизировать потерю стиля.  
Обратите внимание: вам нужно передать функцию, которая вычисляет потерю, и функцию, которая вычисляет градиенты, как два отдельных аргумента

На рис. 8.8 показано, что получается в результате. Имейте в виду, что этот прием — лишь одна из форм ретекстурирования изображений, или передачи текстуры. Лучшие результаты с его применением получаются, если изображения с образцами стилей сильно текстурированы и самоподобны, а целевые изображения с содержимым не требуют различия мелких деталей, чтобы их можно было опознать. Этот прием не наделен возможностями абстрагирования — с его помощью едва ли получится перенести стиль с одного портрета в другой. Данный алгоритм ближе к классической обработке сигналов, чем к ИИ, поэтому не нужно ожидать от него чего-то сверхъестественного!

Кроме того, учтите, что этот алгоритм передачи стиля выполняется довольно медленно. Однако выполняемые преобразования достаточно просты, чтобы их можно было исследовать с использованием небольшой и быстрой сверточной сети при наличии достаточного объема обучающих данных. Быстрой передачи стиля можно достичь, если сначала потратить много времени на создание входных/выходных обучающих примеров для фиксированного изображения с образцом стиля, используя метод, описанный здесь, а затем обучить простую сверточную сеть данному конкретному преобразованию стиля. После этого можно будет почти мгновенно



**Рис. 8.8.** Несколько примеров результатов

стилизовать любое изображение: для этого потребуется просто пропустить его через эту маленькую сверточную сеть.

### 8.3.4. Подведение итогов

- ❑ Передача стиля заключается в создании нового изображения, которое сохраняет содержимое целевого изображения и оформлено в стиле изображения-образца.
- ❑ Содержимое может сохраняться активациями верхнего слоя сверточной сети.
- ❑ Стиль может сохраняться внутренними корреляциями активаций разных слоев.
- ❑ Таким образом, передачу стиля в глубоком обучении можно сформулировать как процесс оптимизации, использующий функцию потерь, которая определяется предварительно обученной сверточной сетью.
- ❑ Начав с этой простой идеи, можно реализовать множество разнообразных вариантов.

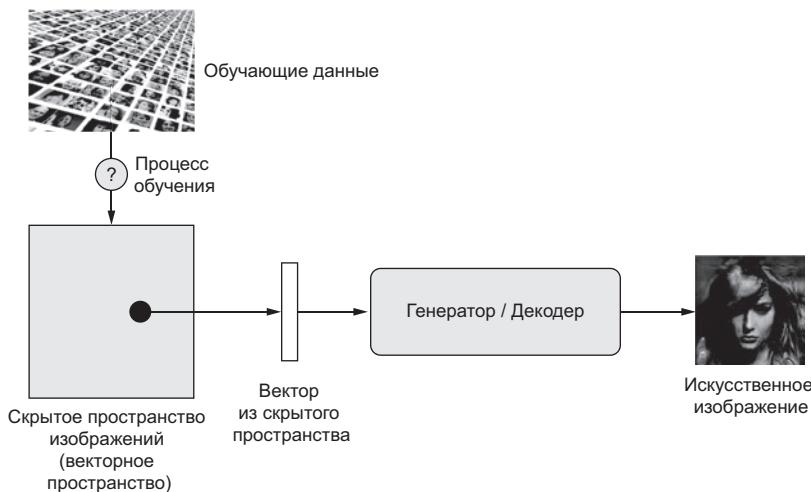
## 8.4. Генерирование изображений с вариационными автокодировщиками

Выбор шаблонов из скрытого пространства изображений для создания совершенно новых или редактирования существующих изображений в настоящее время является самым успешным и популярным применением художественных возможностей ИИ. В этом и в следующем разделах мы рассмотрим некоторые высококачественные понятия, связанные с генерацией изображений, а также детали реализации двух основных методов, используемых в этой области: *вариационных автокодировщиков* (Variational AutoEncoders, VAE) и *генеративно-состязательных сетей* (Generative Adversarial Networks, GAN). Приемы, представленные здесь, можно применять не только к изображениям. Используя GAN и VAE, можно создавать скрытые пространства звуков, музыки или даже текста, однако на практике наиболее интересные результаты получаются с изображениями, и именно поэтому мы сосредоточимся на этом направлении.

### 8.4.1. Выбор шаблонов из скрытых пространств изображений

Основная идея генерации изображений заключается в создании малоразмерного скрытого пространства представлений (которое, естественно, является пространством векторов), любая точка которого может отображаться в реалистично выглядящее изображение. Модуль, способный реализовать это отображение, который принимает на входе скрытую точку и выводит изображение (сетку пикселов),

называют генератором (в случае использования GAN) или декодером (если используется VAE). Создав скрытое пространство, вы сможете выбирать точки из него, целенаправленно или произвольно, и отображать их в пространство изображений, генерируя изображения, не встречавшиеся прежде (рис. 8.9).



**Рис. 8.9.** Обученное скрытое векторное пространство изображений и его использование для создания новых изображений



**Рис. 8.10.** Непрерывное пространство лиц, сгенерированное Томом Уайтом (Tom White) с использованием VAE

Генеративно-состязательные сети и вариационные автокодировщики — это две разные стратегии получения таких скрытых пространств из изображений, и каждая из них имеет свои отличительные характеристики. Вариационные автокодировщики прекрасно подходят для получения хорошо структурированных скрытых пространств, когда конкретные направления кодируют значимые оси изменений в данных. Генеративно-состязательные сети генерируют изображения, потенциально более реалистичные, но скрытое пространство, из которого они исходят, может не обладать структурированностью и непрерывностью.

## 8.4.2. Концептуальные векторы для редактирования изображений

Мы уже обсуждали идею *концептуальных векторов*, когда рассматривали векторные представления слов в главе 6. Сама идея остается прежней: некоторые направления в скрытом (векторном) пространстве представлений могут кодировать интересные оси изменений исходных данных. Например, в скрытом пространстве изображений лиц может иметься вектор «улыбка»  $s$  такой, что если скрытая точка  $z$  является векторным представлением некоторого лица, тогда точка  $z + s$  является векторным представлением того же лица, но улыбающегося. После выявления такого вектора становится возможным редактировать изображения, проецируя их в скрытое пространство, перемещая представления значимым способом и декодируя их обратно в пространство изображений. Концептуальные векторы существуют для практически любых независимых вариаций в пространстве изображений. В случае с лицами можно обнаружить векторы, добавляющие солнцезащитные очки, удаляющие очки, преобразующие мужское лицо в женское, и т. д. На рис. 8.11 приводится пример вектора «улыбка», концептуального вектора, обнаруженного Томом Уайтом (Tom White) из школы дизайна в университете Виктории (Victoria University School of Design) в Новой Зеландии, с использованием VAE, обученных на наборах изображений лиц знаменитостей (набор данных CelebA).

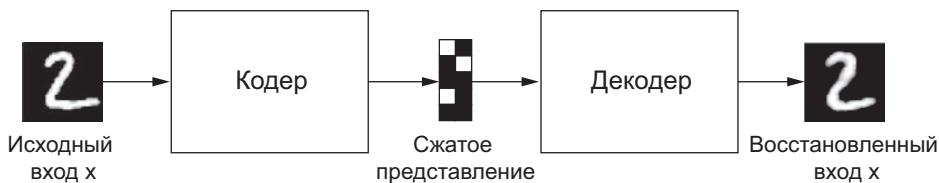


**Рис. 8.11.** Вектор «улыбка»

### 8.4.3. Вариационные автокодировщики

Вариационные автокодировщики, открытые одновременно Дидериком Кингма (Diederik P. Kingma) и Максом Веллингом (Max Welling) в декабре 2013 года<sup>1</sup> и Данило Йименезом Резенде (Danilo Jimenez Rezende), Шакиром Мохамедом (Shakir Mohamed) и Дааном Вирстрой (Daan Wierstra) в январе 2014 года<sup>2</sup>, являются разновидностью генеративной модели, которая особенно хорошо подходит для редактирования изображений посредством концептуальных векторов. Они представляют собой современный подход к автокодировщикам — разновидности сетей, целью которых является кодирование входного малоразмерного скрытого пространства и последующего его декодирования, — которые сочетают идеи из глубокого обучения и байесовского вывода.

Классический автокодировщик изображений принимает изображение, отображает его в скрытое векторное пространство с помощью модуля кодирования и декодирует его обратно в выходное изображение с теми же размерами с помощью модуля декодирования (рис. 8.12). Затем он обучается, используя в качестве целевых данных те же изображения, что подавались на вход. Таким образом, автокодировщик учится восстанавливать исходные данные. Накладывая различные ограничения на код (вывод кодировщика), можно получить автокодировщик, чтобы извлечь из данных более или менее интересные скрытые представления. Чаще всего код ограничивается малым числом измерений и разреженностью (когда большинство элементов имеют нулевые значения). В этом случае кодировщик действует как инструмент сжатия входных данных, генерируя на выходе меньший объем информации.



**Рис. 8.12.** Автокодировщик: отображает вход  $x$  в сжатое представление, которое затем декодируется обратно как  $x'$

На практике такие классические автокодировщики не создают особенно полезных или хорошо структурированных скрытых пространств. Также они не очень хороши как инструмент сжатия. По этой причине они почти вышли из моды. Вариационные автокодировщики, однако, добавляют в автокодировщики толику статистического

<sup>1</sup> Diederik P. Kingma и Max Welling, «Auto-Encoding Variational Bayes», arXiv (2013), <https://arxiv.org/abs/1312.6114>.

<sup>2</sup> Danilo Jimenez Rezende, Shakir Mohamed и Daan Wierstra, «Stochastic Backpropagation and Approximate Inference in Deep Generative Models», arXiv (2014), <https://arxiv.org/abs/1401.4082>.

волшебства, что заставляет их извлекать непрерывные, высокоструктурированные скрытые пространства. Они оказались мощным инструментом для генерирования изображений.

Вместо сжатия в фиксированный код в скрытом пространстве вариационный кодировщик превращает входное изображение в параметры статистического распределения: среднее и дисперсию. По сути, это означает предположение, что входное изображение было сгенерировано статистическим процессом и что случайную составляющую этого процесса необходимо учитывать в ходе кодирования и декодирования. Вариационный автокодировщик затем использует среднее и дисперсию как параметры для случайного отбора одного элемента из распределения и декодирует его обратно в оригинал изображение (рис. 8.13). Стохастичность этого процесса повышает надежность и заставляет скрытое пространство кодировать значимые представления: каждая точка, выбранная в скрытом пространстве, декодируется в допустимый вывод.



**Рис. 8.13.** Вариационный автокодировщик отображает изображение в два вектора,  $z_{\text{mean}}$  и  $z_{\log \text{sigma}}$ , которые определяют распределение вероятности в скрытом пространстве, используемом для выбора точки для декодирования

Вот как работает вариационный автокодировщик с технической точки зрения:

- Модуль кодирования превращает выборки из входного изображения `input_img` в два параметра в скрытом пространстве, `z_mean` и `z_log_variance`.
- Вы выбираете из скрытого нормального распределения произвольную точку `z` для генерации входного изображения как  $z = z_{\text{mean}} + \exp(z_{\log \text{variance}}) * \epsilon$ , где `epsilon` — это случайный тензор небольших значений.
- Модуль декодера отображает эту точку из скрытого пространства обратно в оригинал изображение.

Поскольку `epsilon` является случайным тензором, процесс гарантирует, что каждая точка, близкая к скрытому местоположению, где закодировано `input_img` ( $z$ -mean), может быть декодирована в нечто, похожее на `input_img`, что обеспечивает непрерывную значимость скрытого пространства. Любые две близкие точки в скрытом пространстве будут декодированы в очень похожие изображения. Непрерывность в сочетании с малой размерностью скрытого пространства заставляет каждое направление в скрытом пространстве кодировать значимую ось изменений данных, что делает скрытое пространство высокоструктурированным и прекрасно подходящим для манипуляций посредством концептуальных векторов.

Параметры вариационного автокодировщика обучаются на двух функциях потерь: *потерях восстановления* (reconstruction loss), которая заставляет декодированные образцы совпадать с исходными входами, и *потерях регуляризации* (regularization loss), которая помогает извлекать хорошо сформированные скрытые пространства и ослабляет проблему переобучения на обучающих данных. Давайте пройдемся по реализации вариационного автокодировщика в Keras. Схематически она выглядит так:

```

z_mean, z_log_variance = encoder(input_img)           ← Кодирование входа
                                                               в среднее и дисперсию
→ z = z_mean + exp(z_log_variance) * epsilon
reconstructed_img = decoder(z) ← Декодирование z обратно в изображение
model = Model(input_img, reconstructed_img)           ← Создание модели автокодировщика, которая отображает
                                                               входное изображение в его
                                                               реконструкцию
Извлечение скрытой точки с использованием
случайной величины epsilon
  
```

Затем можно обучить модель, используя потери восстановления и потери регуляризации.

В следующем листинге демонстрируется используемая нами сеть кодировщика, отображающая изображения в параметры распределения вероятности в скрытом пространстве. Эта простая сверточная сеть отображает входное изображение  $x$  в два вектора,  $z\_mean$  и  $z\_log\_var$ .

### Листинг 8.23. Сеть кодировщика VAE

```

import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

img_shape = (28, 28, 1)
batch_size = 16
  
```

### **Листинг 8.23** (продолжение)

`latent_dim = 2` ← Размерность скрытого пространства: двумерная плоскость

```
input_img = keras.Input(shape=img_shape)
x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x) | Входное изображение кодируется
z_log_var = layers.Dense(latent_dim)(x) | в эти два параметра
```

Далее приводится код, использующий `z_mean` и `z_log_var`, параметры статистического распределения, которое, как предполагается, произвело `input_img`, для создания точки `z` скрытого пространства. Здесь мы обернули некоторый произвольный код (основанный на примитивах Keras) в слой `Lambda`. В Keras все сущее должно заключаться в слои, поэтому код, не являющийся частью встроенного слоя, необходимо заключать в слой `Lambda` (или в свой собственный слой).

**Листинг 8.24.** Функция выбора точки из скрытого пространства

```

def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                               mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])

```

Следующий листинг демонстрирует реализацию декодера. Здесь мы приводим размерность вектора  $z$  в соответствие с размерами изображения и затем используем несколько сверточных слоев, чтобы получить выходное изображение с теми же размерами, что и оригинальное `input_img`.

**Листинг 8.25.** Сеть декодера VAE, отображающая точки из скрытого пространства в изображения

decoder\_input = layers.Input(K.int\_shape(z)[1:])  $\leftarrow$  Передача  $z$  на вход

```
→ x = layers.Reshape(shape_before_flattening[1:])(x)

x = layers.Conv2DTranspose(32, 3,
                           padding='same',
                           activation='relu',
                           strides=(2, 2))(x)

x = layers.Conv2D(1, 3,
                  padding='same',
                  activation='sigmoid')(x)

decoder = Model(decoder_input, x)

z_decoded = decoder(z) ←

Применяет
декодер к z, чтобы
восстановить
декодированное
значение z
```

Преобразование  $z$  в карту признаков с той же формой, которую имела карта признаков перед последним слоем Flatten в модели кодировщика

Двойственная природа потерь VAE не соответствует традиционной форме `loss(input, target)`. Поэтому мы реализуем вычисление потерь, написав свой слой, который внутренне использует встроенный метод `add_loss` слоя для создания произвольных потерь.

**Листинг 8.26.** Собственный слой для вычисления потерь VAE

```
class CustomVariationalLayer(keras.layers.Layer):  
  
    def vae_loss(self, x, z_decoded):  
        x = K.flatten(x)  
        z_decoded = K.flatten(z_decoded)  
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)  
        kl_loss = -5e-4 * K.mean(  
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)  
        return K.mean(xent_loss + kl_loss)  
  
    def call(self, inputs): ← Собственные слои реализуются  
        x = inputs[0] | определением метода call  
        z_decoded = inputs[1]  
        loss = self.vae_loss(x, z_decoded)  
        self.add_loss(loss, inputs=inputs)  
        return x  
  
y = CustomVariationalLayer()([input_img, z_decoded]) ←  
  
мы не используем этот  
результат, однако слой должен  
что-то возвращать  
  
Вызов собственного слоя  
с исходными и декодированными  
данными для получения  
окончательного вывода модели
```

Наконец, мы готовы создать и обучить модель. Поскольку вычислением потерь у нас занимается собственный слой, мы не указываем функцию потерь на этапе компиляции (`loss=None`). Это, в свою очередь, означает, что нам не нужно передавать целевые данные в процесс обучения (как можно заметить, в метод `fit` обучаемой модели передается только `x_train`).

### Листинг 8.27. Обучение VAE

```
from keras.datasets import mnist

vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))
```

После обучения такой модели — в данном случае на наборе MNIST — мы можем использовать сеть `decoder` для превращения произвольных векторов из скрытого пространства в изображения.

### Листинг 8.28. Выбор сетки с точками из двумерного скрытого пространства и их декодирование в изображения

```
import matplotlib.pyplot as plt
from scipy.stats import norm

n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
```

`n = 15`

`digit_size = 28`

`figure = np.zeros((digit_size * n, digit_size * n))`

`grid_x = norm.ppf(np.linspace(0.05, 0.95, n))`

`grid_y = norm.ppf(np.linspace(0.05, 0.95, n))`

Будет отображаться сетка  $15 \times 15$  цифр (всего 225 цифр)

Преобразует координаты линейного пространства с использованием функции `ppf` из пакета SciPy для получения значений скрытой переменной `z` (поскольку предшествующее скрытое пространство является гауссовым)

Многократное повторение выбора `z` для формирования полного пакета

```

x_decoded = decoder.predict(z_sample, batch_size=batch_size)
digit = x_decoded[0].reshape(digit_size, digit_size)
figure[i * digit_size: (i + 1) * digit_size,
       j * digit_size: (j + 1) * digit_size] = digit

```

```

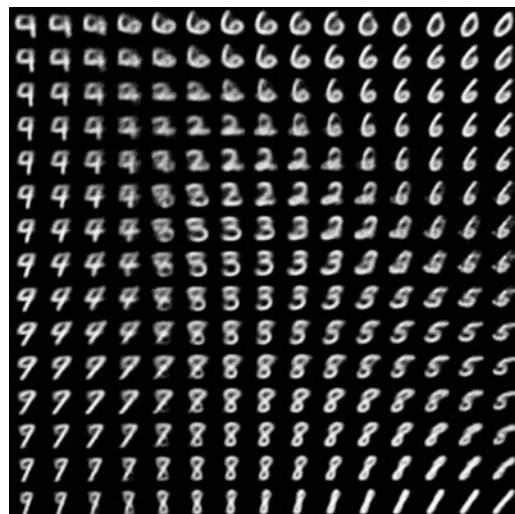
plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()

```

Преобразование первой цифры  
в пакете из размерности  $28 \times 28 \times 1$   
в  $28 \times 28$

Декодирование пакета  
в изображения цифр

Сетка выбранных цифр (рис. 8.14) демонстрирует полностью непрерывное распределение разных классов цифр, где одна цифра превращается в другую по пути через непрерывное скрытое пространство. Конкретные направления в этом пространстве наделены определенным смыслом: например, есть направление «четверочности», «единичности» и т. д.



**Рис. 8.14.** Сетка с цифрами, декодированными из скрытого пространства

В следующем разделе мы подробно рассмотрим один важный инструмент создания искусственных изображений: генеративно-состязательные сети (Generative Adversarial Networks, GAN).

#### 8.4.4. Подведение итогов

- Генерирование изображений с применением глубокого обучения происходит за счет выделения скрытых пространств, несущих статистическую информацию о наборе изображений. Выбирая точки из скрытого пространства и декодируя их, можно видеть прежде не встречавшиеся изображения. Существует два основных инструмента для решения этой задачи: вариационные автокодировщики (VAE) и генеративно-состязательные сети (GAN).
- Вариационные автокодировщики создают структурированные, непрерывные скрытые представления. По этой причине они хорошо подходят для любых видов редактирования изображений в скрытом пространстве: подмена лица, превращение нахмуренного лица в улыбающееся и т. д. Они также хорошо подходят для создания мультиплексации путем прохождения через раздел скрытого пространства, когда начальное изображение постепенно и непрерывно преобразуется в другие изображения.
- Генеративно-состязательные сети позволяют генерировать реалистичные однокадровые изображения, однако они не порождают скрытых пространств, непрерывных и с четкой структурой.

Большинство успешных практических применений в области графики, которые мне приходилось видеть, основаны на вариационных автокодировщиках, а генеративно-состязательные сети пользуются очень большой популярностью в академической среде, по крайней мере так было в 2016–2017 годах. Как они действуют и как реализуются, вы узнаете в следующем разделе.

#### СОВЕТ

Для экспериментов с генерацией изображений я рекомендую использовать набор Large-scale Celeb Faces Attributes (CelebA). Этот набор доступен для загрузки бесплатно и содержит более 200 000 портретов знаменитостей. В частности, он прекрасно подходит для экспериментов с концептуальными векторами и в этом смысле превосходит набор MNIST.

### 8.5. Введение в генеративно-состязательные сети

Генеративно-состязательные сети (Generative Adversarial Networks, GAN), впервые представленные в 2014 году Яном Гудфеллоу (Ian Goodfellow) и его коллегами<sup>1</sup>, —

<sup>1</sup> Ian Goodfellow et al., «Generative Adversarial Networks», arXiv (2014), <https://arxiv.org/abs/1406.2661>.

альтернатива вариационным автокодировщикам, позволяющая выделять скрытые пространства изображений. Они позволяют генерировать очень реалистичные искусственные изображения, статистически неотличимые от настоящих.

Чтобы проще было понять суть генеративно-состязательной сети, вообразите фальсификатора, пытающегося подделать картину Пикассо. Сначала он довольно плохо справляется с задачей. Он показывает свои подделки вместе с подлинниками Пикассо продавцу произведений искусства. Продавец оценивает подлинность картин и рассказывает фальсификатору, какие детали делают картину похожей на картину Пикассо. Фальсификатор возвращается в мастерскую и создает несколько новых подделок. С течением времени фальсификатор становится все более компетентным в имитации стиля Пикассо, а продавец — все более опытным в различении подделок. В конце концов у них на руках оказываются превосходные подделки Пикассо.

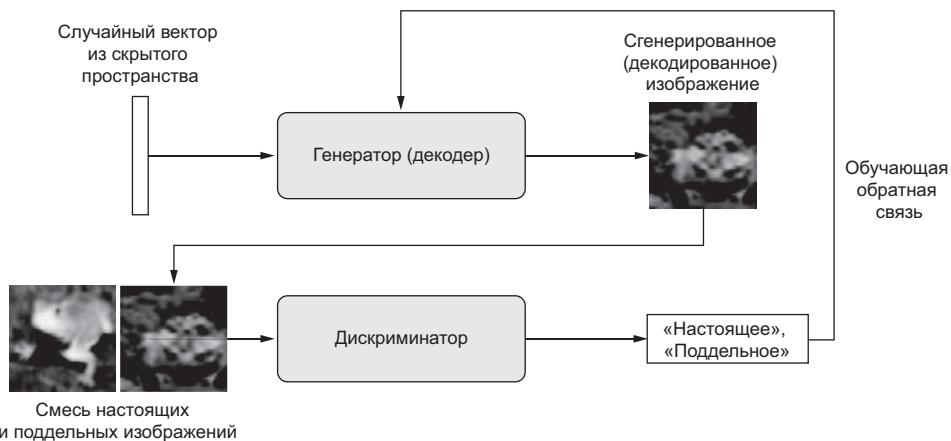
Вот что такое генеративно-состязательная сеть: она состоит из двух сетей — выполняющей подделку и оценивающей эту подделку, постепенно обучающих друг друга:

- *сеть-генератор* — получает на входе случайный вектор (случайную точку в скрытом пространстве) и декодирует его в искусственное изображение;
- *сеть-дискриминатор* (или *противник*) — получает изображение (настоящее или поддельное) и определяет, взято ли это изображение из обучающего набора или сгенерировано сетью-генератором.

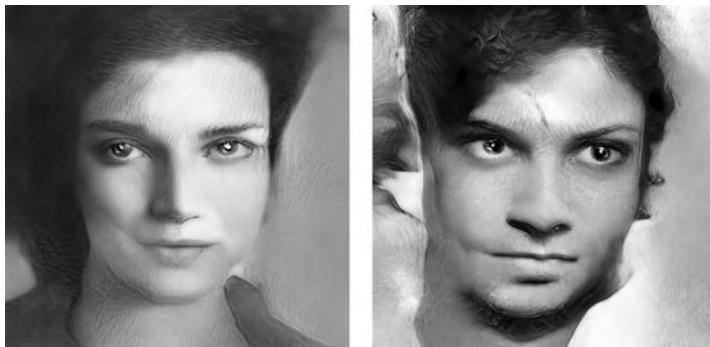
Сеть-генератор обучается обманывать сеть-дискриминатор и, соответственно, учится создавать все более реалистичные изображения: поддельные изображения, неотличимые от настоящих в той мере, на какую способна сеть-дискриминатор (рис. 8.15). Сеть-дискриминатор, в свою очередь, постоянно адаптируется к увеличивающейся способности сети-генератора и устанавливает все более высокую планку реализма для генерируемых изображений. По окончании обучения генератор способен превратить любую точку из своего входного пространства в правдоподобное изображение. В отличие от вариационных автокодировщиков, это скрытое пространство дает меньше гарантий наличия в нем значимой структуры; в частности, оно не является непрерывным.

Примечательно, что генеративно-состязательная сеть (GAN) — это система, в которой минимум оптимизации не фиксирован, в отличие от любых других обучаемых конфигураций, которые вы могли видеть в этой книге. Обычно градиентный спуск заключается в постепенном скатывании вниз по холмам статического ландшафта потерь. Однако в случае с GAN каждый шаг вниз по склону немного меняет весь ландшафт. Это динамическая система, в которой процесс оптимизации стремится не к минимуму, а к равновесию двух сил. По этой причине генеративно-состязательные сети трудно поддаются обучению: чтобы получить действующую генера-

тивно-состязательную сеть, требуется приложить большие усилия по настройке архитектуры модели и параметров обучения.



**Рис. 8.15.** Генератор преобразует случайные скрытые векторы в изображения, а дискриминатор стремится отличить настоящие изображения от сгенерированных искусственно. Генератор обучается обманывать дискриминатор



**Рис. 8.16.** Скрытое пространство жителей. Изображения сгенерированы Майком Тикой (Mike Tyka) с использованием многоступенчатой генеративно-состязательной сети, обученной на наборе изображений лиц ([www.miketyka.com](http://www.miketyka.com))

### 8.5.1. Реализация простейшей генеративно-состязательной сети

В этом разделе я расскажу, как реализовать простейшую генеративно-состязательную сеть с использованием Keras, потому что сети этого вида очень сложны, а подробное описание технических деталей выходит далеко за рамки этой книги.

Данная реализация — *глубокая сверточная генеративно-состязательная сеть* (Deep Convolutional GAN, DCGAN), в которой генератор и дискриминатор являются глубокими сверточными сетями. В ней, например, используется слой `Conv2DTranspose` для увеличения разрешения изображения в генераторе.

Мы будем обучать GAN на изображениях из набора CIFAR10, содержащего 50 000 изображений  $32 \times 32$  в формате RGB, которые делятся на 10 классов (по 5000 изображений в каждом классе). Для простоты мы используем только изображения, принадлежащие классу «лягушка».

В общих чертах GAN выглядит примерно так:

1. Сеть `generator` отображает векторы с формой (`размерность_скрытого_пространства,`) в изображения с формой (32, 32, 3).
2. Сеть `discriminator` отображает изображения с формой (32, 32, 3) в оценку вероятности того, что изображение является настоящим.
3. Сеть `gan` объединяет генератор и дискриминатор `gan(x) = discriminator(generator(x))`. То есть сеть `gan` отображает скрытое пространство векторов в оценку реализма этих скрытых векторов, декодированных генератором.
4. Мы обучим дискриминатор на примерах реальных и искусственных изображений, отмеченных метками «настоящее»/«поддельное», как самую обычную модель классификации изображений.
5. Для обучения генератора мы используем градиенты весов генератора в отношении потерь модели `gan`. То есть на каждом шаге мы будем смещать веса генератора в направлении увеличения вероятности классификации дискриминатором изображений, декодированных генератором как «настоящие». Иными словами, мы будем обучать генератор обманывать дискриминатор.

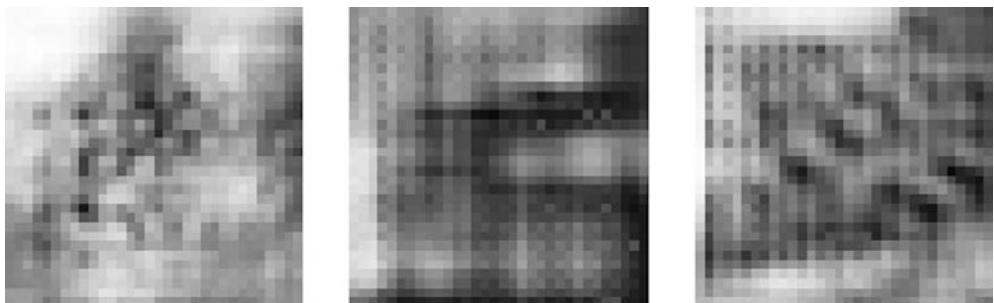
## 8.5.2. Набор хитростей

Процесс обучения и настройки генеративно-состязательных сетей очень сложен. Однако есть несколько хитростей, которые следует знать и помнить. Как и многое другое в глубоком обучении, это больше алхимия, чем наука: все хитрости, описываемые далее, выявлены экспериментальным путем и не имеют теоретического обоснования. Они опираются на интуитивное понимание явления и хорошо работают на практике, хотя и не во всех контекстах.

Вот несколько хитростей, используемых в реализации генератора и дискриминатора GAN в этом разделе. Это не полный список; еще множество хитростей, имеющих отношение к GAN, можно найти в специализированной литературе:

- ❑ В качестве последней функции активации в генераторе мы используем `tanh` вместо `sigmoid`, которую часто можно встретить в моделях других типов.

- ❑ Мы будем выбирать точки из скрытого пространства, используя *нормальное распределение* (распределение Гаусса), а не равномерное.
- ❑ Стохастичность повышает устойчивость. Поскольку целью обучения является динамическое равновесие, генеративно-состязательные сети легко могут застревать на разных препятствиях. Введение случайной составляющей в процесс обучения помогает предотвратить это. Мы вводим случайный компонент двумя способами: используя прореживание в дискриминаторе и добавляя случайный шум в метки для дискриминатора.
- ❑ Разреженные градиенты могут препятствовать обучению GAN. В глубоком обучении разреженность часто является желательным свойством, но не в случае с GAN. Разреженность градиента могут вызывать: операции выбора максимального значения по соседним элементам (max pooling) и активации ReLU. Вместо выбора максимального значения для уменьшения разрешения мы рекомендуем использовать чередующиеся свертки, а вместо функции активации ReLU — слой LeakyReLU. Он напоминает ReLU, но ослабляет ограничение разреженности, допуская небольшие отрицательные значения активации.
- ❑ В сгенерированных изображениях часто наблюдаются артефакты типа «шахматная доска», обусловленные неравномерным охватом пространства пикселов в генераторе (рис. 8.17). Для их устранения мы будем выбирать размер ядра, кратный размеру шага, при каждом использовании разреженных слоев Conv2DTranspose или Conv2D в генераторе и дискриминаторе.



**Рис. 8.17.** Артефакты типа «шахматная доска», вызванные несовпадением размеров шага и ядра, из-за чего происходит неравномерный охват пространства пикселов: одна из многих тонкостей GAN, доставляющих хлопоты

### 8.5.3. Генератор

Сначала реализуем модель **generator**, которая преобразует вектор (из скрытого пространства, полученного во время обучения, который будет выбираться случай-

но) в изображение-кандидат. Одна из многих проблем, которые часто возникают в сетях GAN, — генератор создает изображения, которые выглядят как шум. Одно из возможных решений — использовать прореживание в дискриминаторе и генераторе.

#### Листинг 8.29. Сеть генератора в GAN

```

import keras
from keras import layers
import numpy as np

latent_dim = 32
height = 32
width = 32
channels = 3

generator_input = keras.Input(shape=(latent_dim,))

x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))           | Преобразование входа
                                              | в карту признаков  $16 \times 16$ 
                                              | со 128 каналами

x = layers.Conv2D(256, 5, padding='same')(x)   | Увеличение разрешения до  $32 \times 32$ 
x = layers.LeakyReLU()(x)

x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()                         | Производит карту признаков
                                              |  $32 \times 32$  с 1 каналом
                                              | (форма изображений в наборе CIFAR10)

→ Создание модели генератора, которая
отображает вход с формой (размерность_
скрытого_пространства,) в изображение
с формой ( $32, 32, 3$ )                                |
                                              | (форма изображений в наборе CIFAR10)

```

#### 8.5.4. Дискриминатор

Теперь перейдем к модели `discriminator`, которая принимает на входе изображение-кандидат (реальное или искусственное) и относит его к одному из двух классов: «подделка» или «настоящее, имеющееся в обучающем наборе».

**Листинг 8.30.** Сеть дискриминатора в GAN

```

discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)

x = layers.Dropout(0.4)(x) ←———— Уровень прореживания: важная хитрость!

x = layers.Dense(1, activation='sigmoid')(x) ←———— Уровень классификации

discriminator = keras.models.Model(discriminator_input, x) ←————
discriminator.summary()

discriminator_optimizer = keras.optimizers.RMSprop(
    lr=0.0008,
    clipvalue=1.0, ←———— Использование градиентной обрезки (по значению) в оптимизаторе
    decay=1e-8) ←———— Для стабилизации используется затухание скорости обучения

discriminator.compile(optimizer=discriminator_optimizer,
                      loss='binary_crossentropy')

```

Создание модели дискриминатора, которая преобразует вход с формой (32, 32, 3) в бинарное решение (подделка/настоящее)

### 8.5.5. Состязательная сеть

Наконец, перейдем к состязательной сети, объединяющей генератор и дискриминатор. В процессе обучения эта модель будет смешивать веса генератора в направлении увеличения способности обмана дискриминатора. Эта модель преобразует точки скрытого пространства в классифицирующее решение — «подделка» или «настоящее» — и предназначена для обучения с метками, которые всегда говорят: «это настоящие изображения». То есть обучение `gan` будет смешивать веса в модели `generator` так, чтобы увеличить вероятность получить от дискриминатора ответ «настоящее», когда тот будет просматривать поддельное изображение. Важно также отметить, что дискриминатор нужно «заморозить» на время обучения (отключить его обучение): его веса не должны обновляться при обучении `gan`. В противном случае все сведется к тому, что вы обучите дискриминатор всегда отвечать «настоящее», а это едва ли вам нужно!

**Листинг 8.31.** Состязательная сеть

```

discriminator.trainable = False
gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')

```

**8.5.6. Как обучить сеть DCGAN**

Теперь можно приступать к обучению. Ниже схематически описывается общий цикл обучения. В каждой эпохе нужно выполнить следующие действия:

1. Извлечь случайные точки из скрытого пространства (случайный шум).
2. Создать изображения с помощью генератора, используя случайный шум.
3. Смешать сгенерированные изображения с настоящими.
4. Обучить дискриминатор на этом смешанном наборе изображений, добавив соответствующие цели: «настоящее» (для настоящих изображений) или «подделка» (для сгенерированных изображений).
5. Выбрать новые случайные точки из скрытого пространства.
6. Обучить `gan`, используя эти случайные векторы, с целями, которые всегда говорят: «это настоящие изображения». Это приведет к смещению весов генератора (и только генератора, потому что внутри `gan` дискриминатор «замораживается») в направлении, увеличивающем вероятность получить от дискриминатора ответ «настоящее» для сгенерированных изображений: это научит генератор обманывать дискриминатор.

Реализуем эту последовательность.

**Листинг 8.32.** Реализация обучения GAN

```

import os
from keras.preprocessing import image
(x_train, y_train), (_, _) = keras.datasets.cifar10.load_data() ← Загрузка данных CIFAR10
x_train = x_train[y_train.flatten() == 6] ← Выбирает изображения лягушек (класс 6)
x_train = x_train.reshape(
    (x_train.shape[0],) +
    (height, width, channels)).astype('float32') / 255. ← Нормализация данных

```

Продолжение ↗

Листинг 8.32 (продолжение)

```

iterations = 10000           Каталог для сохранения
batch_size = 20               сгенерированных
save_dir = 'your_dir'         изображений

start = 0
for step in range(iterations):
    random_latent_vectors = np.random.normal(size=(batch_size,
                                                    latent_dim))

generated_images = generator.predict(random_latent_vectors)      ←
                                                                    Декодирование их в под-
                                                                    дельные изображения

stop = start + batch_size
real_images = x_train[start: stop]
combined_images = np.concatenate([generated_images, real_images])   ←

Объединение их с настоящими
изображениями
labels = np.concatenate([np.ones((batch_size, 1)),
                        np.zeros((batch_size, 1))])   ← Сборка меток, отли-
                                                                    чающихся настоящие
labels += 0.05 * np.random.uniform(labels.shape)                   изображения от под-
                                                                    дельных

Добавление случайного шума в метки — важная хитрость!
d_loss = discriminator.train_on_batch(combined_images, labels)   ←
                                                                    Обучение дискриминатора

random_latent_vectors = np.random.normal(size=(batch_size,
                                                latent_dim))          ← Выбор
                                                                    случайных
                                                                    точек из
                                                                    скрытого
                                                                    простран-
                                                                    ства

misleading_targets = np.zeros((batch_size, 1)) ///
a_loss = gan.train_on_batch(random_latent_vectors,             ←
                            misleading_targets)           ← Обучение генератора
                                                                    (через модель gan,
                                                                    которая замораживает
                                                                    веса дискриминатора)

start += batch_size
if start > len(x_train) - batch_size:
    start = 0

if step % 100 == 0:   ← Сохранение изображений (через каждые 100 шагов)
    gan.save_weights('gan.h5')   ← Сохранение весов модели

    print('discriminator loss:', d_loss) | Вывод метрик
    print('adversarial loss:', a_loss)

    img = image.array_to_img(generated_images[0] * 255., scale=False)
    img.save(os.path.join(save_dir,
                          'generated_frog' + str(step) + '.png'))   ←

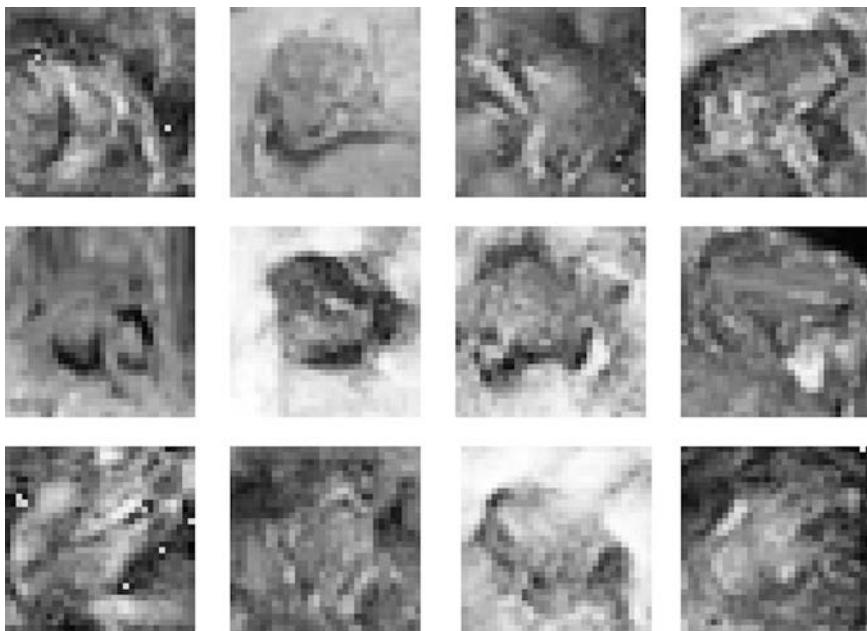
    img = image.array_to_img(real_images[0] * 255., scale=False)
    img.save(os.path.join(save_dir,
                          'real_frog' + str(step) + '.png'))   ←

Сборка меток, которые всегда говорят: «это
настоящие изображения» (это ложь!)   ← Сохранение одного настоящего
                                            изображения для сравнения

                                            Сохранение одного сгенерированного изображения

```

В ходе обучения можно заметить, что потери состязательной сети значительно возрастают, а потери дискриминатора стремятся к нулю — дискриминатор может оказаться в доминирующем положении над генератором. В этом случае попробуйте уменьшить скорость обучения дискриминатора и увеличить его коэффициент прореживания.



**Рис. 8.18.** Работа дискриминатора: в каждом ряду два изображения — поддельные, сгенерированные сетью GAN, и одно настоящее, взятое из обучающего набора. Сможете отличить их? (Настоящие изображения по колонкам, слева направо: в середине, вверху, внизу и в середине.)

### 8.5.7. Подведение итогов

- Генеративно-состязательная сеть состоит из двух сетей: генератора и дискриминатора. Дискриминатор обучается отличать изображения, созданные генератором, от настоящих, имеющихся в обучающем наборе, а генератор обучается обманывать дискриминатор. Примечательно, что генератор вообще не видит изображений из обучающего набора; вся информация, которую он имеет, поступает из дискриминатора.
- Генеративно-состязательные сети сложны в обучении, потому что обучение GAN — это динамический процесс, отличный от обычного процесса градиентного спуска по фиксированному ландшафту потерь. Для правильного обучения

GAN приходится использовать ряд эвристических трюков, а также уделять большое внимание настройкам.

- ❑ Генеративно-состязательные сети потенциально способны производить очень реалистичные изображения. Однако в отличие от вариационных автокодировщиков получаемое ими скрытое пространство не имеет четко выраженной непрерывной структуры, и поэтому они могут не подходить для некоторых практических применений, таких как редактирование изображений с использованием концептуальных векторов.

## Краткие итоги главы

- ❑ Благодаря применению глубокого обучения в творчестве, глубокие нейронные сети вышли за рамки аннотирования существующего и начали генерировать свой контент. В этой главе вы узнали:
  - Как генерировать последовательности данных по одному элементу за раз. Эту методику можно использовать для создания текстов, а также мелодий, нота за нотой, и любых других подобных временных последовательностей.
  - Как работает алгоритм DeepDream: путем максимизации активаций слоя сверточной сети через градиентное восхождение во входном пространстве.
  - Как реализовать передачу стиля, когда объединяются изображения с целявым контентом и образцом стиля и получаются интересные результаты.
  - Что такое генеративно-состязательные сети и вариационные автокодировщики, как их использовать для получения новых изображений и как можно использовать концептуальные векторы из скрытого пространства для редактирования изображений.
- ❑ Эти несколько методов охватывают лишь самые основы этого быстроразвивающегося направления. Вам еще многое предстоит узнать — генеративное глубокое обучение достойно отдельной книги.

# 9

## Заключение

Эта глава охватывает следующие темы:

- ✓ важные уроки этой книги;
- ✓ ограничения глубокого обучения;
- ✓ будущее глубокого обучения, машинного обучения и ИИ;
- ✓ ресурсы для дальнейшего изучения и использования в работе.

Вы почти добрались до конца книги. В этой последней главе обобщаются и повторяются основные понятия, но она также расширит ваши горизонты, позволив выйти за пределы относительно простых понятий, с которыми вы познакомились. Знакомство с глубоким обучением и ИИ — целое путешествие, и конец этой книги — лишь первый шаг на этом пути. Я хочу убедиться, что вы это осознали и хорошо подготовились, чтобы пойти дальше самостоятельно.

Сначала мы окинем взглядом все то, что вы должны вынести из этой книги. Это поможет вам освежить в памяти некоторые понятия, которые вы уже изучили. Затем мы рассмотрим некоторые ключевые ограничения глубокого обучения. Чтобы использовать инструмент правильно, вы должны знать не только его *возможности*, но и его *недостатки*. В заключение я изложу некоторые умозрительные идеи о будущем развитии области глубокого обучения, машинного обучения и искусственного интеллекта (ИИ). Это должно заинтересовать тех, кто захочет заняться фундаментальными исследованиями. В конце главы приводится краткий перечень ресурсов и стратегий для дальнейшего изучения ИИ и получения сведений о новейших достижениях.

## 9.1. Краткий обзор ключевых понятий

Этот раздел обобщает ключевые выводы из этой книги. Если вам потребуется быстро освежить в памяти все, что вы изучили здесь, прочитайте эти несколько страниц.

### 9.1.1. Разные подходы к ИИ

Прежде всего, глубокое обучение не является синонимом ИИ или даже машинного обучения. *Искусственный интеллект* — это давно существующая, широкая область, которую можно определить как «любые попытки автоматизировать когнитивные процессы», иными словами, автоматизировать мысль. Сюда можно отнести и нечто очень простое, такое как электронные таблицы Excel, и очень сложное, как человекоподобные роботы, способные ходить и разговаривать.

*Машинное обучение* — это конкретный раздел ИИ, целью которого является автоматическая разработка программ (называемых *моделями*) исключительно на основе обучающих данных. Этот процесс превращения данных в программу называется *обучением*. Идея машинного обучения зародилась давно, но ее развитие началось только в 1990-х.

Глубокое обучение является одной из многих ветвей машинного обучения, где модели представлены длинными цепочками геометрических функций, применяемых друг за другом. Эти операции организованы в модули, которые называются *слоями* или *уровнями*: модели глубокого обучения обычно формируются как стек слоев или, в более общем смысле, граф слоев. Слои параметризуются *весами*, которые вычисляются в процессе обучения. *Знание* модели хранится в ее весах, а процесс обучения заключается в поиске лучших значений для этих весов.

Несмотря на то что глубокое обучение — лишь один из множества подходов к машинному обучению, оно не равноценно другим подходам. Глубокое обучение — это успешный прорыв. И вот почему.

### 9.1.2. Что делает глубокое обучение особенным среди других подходов к машинному обучению

В течение всего лишь нескольких лет глубокое обучение достигло огромного успеха в решении широкого круга задач, которые прежде воспринимались как очень сложные для компьютеров, особенно в области машинного восприятия: извлечения полезной информации из изображений, видео, звуков и многое другого. При наличии достаточного объема обучающих данных (например, обучающих данных, предварительно классифицированных людьми) из сенсорной информации можно извлечь почти все то же, что может извлечь человек. Поэтому иногда говорят, что глубокое обучение *решило проблему восприятия*, хотя это верно только для очень узкого определения термина *восприятие*.

Благодаря беспрецедентным техническим успехам, глубокое обучение единолично принесло третье и, безусловно, самое долгое лето ИИ: период повышенного интереса, инвестиций и шумихи в области ИИ. Эта книга как раз писалась в середине этого лета. Завершится ли этот период в ближайшем будущем и что случится в конце — это тема для дискуссий. Одно можно сказать наверняка: в отличие от других летних периодов ИИ, глубокое обучение принесло огромные выгоды ряду крупных технологических компаний, позволив распознавать человеческую речь, оказывать интеллектуальную помощь, классифицировать изображения на уровне человека, значительно улучшить машинный перевод и многое другое. Шумиха отступит, однако устойчивое экономическое и технологическое воздействие глубокого обучения останется. В этом смысле глубокое обучение подобно интернету: страсти по нему могут не утихать несколько лет, но в конечном итоге — это серьезная революция, которая изменит нашу экономику и нашу жизнь.

Я с особым оптимизмом отношусь к глубокому обучению, потому что даже если мы не добьемся дальнейшего технического прогресса в следующем десятилетии, развертывание существующих алгоритмов для каждой прикладной задачи станет поворотным моментом для большинства отраслей. Глубокое обучение — это настоящая революция и в настоящее время прогрессирует невероятно быстрыми темпами благодаря все возрастающим инвестициям в ресурсы и людей. С той точки, где я нахожусь, будущее представляется ярким, хотя краткосрочные ожидания кажутся чересчур оптимистичными; развертывание глубокого обучения в полную меру его потенциала займет не меньше десятилетия.

### 9.1.3. Как правильно воспринимать глубокое обучение

Самое удивительное в глубоком обучении — простота реализации. Еще десять лет тому назад никто не предполагал, что мы добьемся таких успехов в решении задач машинного восприятия, использовав простые параметрические модели, обучаемые методом градиентного спуска. Теперь мы знаем: всё, что нам нужно, — это достаточно большие параметрические модели, обученные методом градиентного спуска на достаточно большом количестве примеров. Как однажды сказал Ричард Фейнман (Richard Feynman) о Вселенной: «Она не сложная, просто очень большая»<sup>1</sup>.

В глубоком обучении всё сущее — векторы: всё — *точки в геометрическом пространстве*. Входные данные моделей (текст, изображения и т. д.) и цели сначала векторизуются — превращаются в начальные векторные пространства входных данных и целей. Каждый слой в модели глубокого обучения реализует одно простое геометрическое преобразование данных, проходящих через него. А вся цепочка слоев в модели образует одно сложное геометрическое преобразование, состоящее из последовательности простых. Это сложное преобразование пытается поточечно отобразить входное пространство в целевое. Оно параметризуется весами слоев,

<sup>1</sup> Интервью с Ричардом Фейнманом (Richard Feynman), *The World from Another Point of View*, телевидение Йоркшира, 1972.

которые итеративно обновляются, в зависимости от качества работы модели. Ключевой характеристикой такого геометрического преобразования является *дифференцируемость*, это совершенно необходимо для обучения весов посредством градиентного спуска. Это означает, что геометрическое преобразование входных данных в выходные должно быть гладким и непрерывным, что является существенным ограничением.

Весь процесс применения сложного геометрического преобразования к входным данным можно представить как человека, пытающегося развернуть смятый комок бумаги: этот комок — многообразие входных данных, с которых начинается модель. Каждое движение человека сродни простому геометрическому преобразованию, выполняемому одним слоем. Полная последовательность движений — это сложное преобразование, реализуемое моделью. Модели глубокого обучения — это математические машины, разворачивающие сложное многообразие входных данных с большим количеством измерений.

В этом заключено волшебство глубокого обучения: преобразование смысла в векторы, в геометрические пространства и постепенное изучение сложных геометрических преобразований, отображающих одно пространство в другое. Всё, что вам нужно, — это пространства с достаточно большой размерностью, чтобы полностью охватить отношения, присутствующие в исходных данных.

Всё основано на одной главной идее: смысл *заключается в попарных отношениях* (между словами в языке, между пикселами в изображении и т. д.) и *эти отношения можно оценить функцией расстояния*. Но имейте в виду, что вопрос, реализует ли мозг смысл через геометрические пространства, — это совершенно другое. Векторные пространства эффективны с вычислительной точки зрения, однако не-трудно представить применение других структур данных для интеллекта, например графов. Первоначально нейронные сети возникли из идеи использования графов как способа кодирования смысла, поэтому они и получили название *нейронные сети*; окружающую область исследований обычно называли *коннекционизмом* (*connectionism*). В настоящее время название *нейронные сети* сохраняется исключительно благодаря традиции — это название весьма далеко от истины, потому что они не являются ни нейронными, ни сетями. В частности, нейронные сети едва ли имеют какое-то сходство с мозгом. Более подходящим было бы название *обучаемые многоуровневые представления*, или *обучаемые иерархические представления*, или даже *глубокие дифференцируемые модели*, или *последовательные геометрические преобразования*, чтобы подчеркнуть непрерывность манипуляций с геометрическим пространством.

#### 9.1.4. Ключевые технологии

Технологическая революция, разворачивающаяся на наших глазах, началась не с какого-то одного прорывного изобретения. Как любая другая революция, она явила-лась результатом накопления большого числа благоприятных факторов — сначала

медленно, а потом лавинообразно. В случае с глубоким обучением можно указать на следующие ключевые факторы:

- ❑ Постепенное появление алгоритмических инноваций, с медленным нарастанием в течение двух десятилетий (начиная с алгоритма обратного распространения ошибки), а затем все быстрее и быстрее благодаря увеличению объемов исследований в области глубокого обучения после 2012 года.
- ❑ Доступность больших объемов сенсорных данных. Только благодаря этому мы смогли понять, что все, что нам нужно, — это достаточно большие модели, обученные на достаточно больших объемах данных. Большие объемы данных, в свою очередь, стали побочным продуктом роста потребительского интернета и закона Мура применительно к хранилищам данных.
- ❑ Доступность недорогого вычислительного оборудования с высокой степенью параллелизма, особенно графических процессоров (GPU), производимых компанией NVIDIA, — первые GPU разрабатывались для игровой индустрии, а затем появились чипы, разработанные специально для нужд глубокого обучения. С самого начала глава NVIDIA Жэнь-Сунь Хуанг (Jensen Huang) отметил рост интереса к глубокому обучению и решил сделать ставку на него.
- ❑ Формирование комплексного стека программных технологий, которые сделали эту вычислительную мощь доступной для людей: языка CUDA, а также фреймворков, таких как TensorFlow, автоматически выполняющих дифференцирование, и Keras, обеспечивающих доступность глубокого обучения для многих.

В будущем глубокое обучение будет использоваться не только специалистами — учеными, аспирантами и инженерами академического профиля, но также любыми разработчиками, как это произошло с веб-технологиями. Всем, кому необходимы интеллектуальные приложения: так же как любой компании в наши дни требуется свой веб-сайт, каждому продукту будет нужно интеллектуально осмысливать данные, генерируемые пользователями. Для приближения этого будущего мы должны создавать инструменты, которые делают глубокое обучение радикально простым в использовании и доступным всем, кто имеет базовые навыки программирования. Keras — первый важный шаг в этом направлении.

### 9.1.5. Обобщенный процесс машинного обучения

Хорошо иметь доступ к чрезвычайно мощному инструменту создания моделей, отображающих любое входное в любое целевое пространство, однако не менее сложной частью процесса машинного обучения является всё то, что предшествует проектированию и обучению таких моделей (а для промышленных моделей — еще и все, что происходит потом). Предпосылкой успешного применения машинного обучения является достаточно полное понимание предметной области, чтобы определять, что можно попытаться предсказать, имея текущий набор данных, и как оценивать успех. В этом вам не смогут помочь никакие современные инструменты

вроде Keras и TensorFlow. Вспомним в общих чертах, как выглядит типичный процесс машинного обучения, описанный в главе 4:

1. Определите задачу: какие данные доступны и что требуется предсказать? Может быть, нужно собрать больше данных или нанять людей, которые займутся классификацией обучающего набора данных вручную?
2. Выберите надежную меру успеха в достижении своих целей. Для простых задач это может быть точность предсказания, но во многих случаях требуется использовать более сложные метрики, зависящие от предметной области.
3. Подготовьте процедуру проверки для оценки моделей. В частности, нужно определить обучающий, проверочный и контрольный наборы данных. Информация из проверочного и контрольного наборов данных не должна просачиваться в обучающие данные: например, в случае с временными последовательностями проверочные и контрольные данные должны следовать непосредственно за обучающими данными.
4. Преобразуйте данные в векторы и выполните предварительную обработку, чтобы сделать их более доступными для нейронной сети (нормализация и т. д.).
5. Реализуйте первую модель, преодолевающую планку базового решения, чтобы убедиться в применимости машинного обучения к данной задаче. Так бывает не всегда!
6. Постепенно совершенствуйте архитектуру своей модели, настраивая гиперпараметры и добавляя регуляризацию. Вносите изменения для увеличения качества, опираясь только на проверочные данные, — ни контрольные, ни обучающие данные не должны учитываться на этом этапе. Помните, что вы должны довести свою модель до состояния переобучения (чтобы определить уровень мощности модели, покрывающей ваши потребности) и только потом добавлять регуляризацию или уменьшать размер модели.
7. Помните о переобучении на проверочном наборе данных, выполняя настройку гиперпараметров: гиперпараметры могут оказаться чрезмерно специализированными для проверочного набора. Чтобы избежать этого, создайте отдельный контрольный набор!

### 9.1.6. Основные архитектуры сетей

Есть три семейства архитектур сетей, которые вы должны знать: *полносвязные*, *сверточные* и *рекуррентные сети*. Каждый тип сетей предназначен для конкретной модальности входных данных: архитектура сети (полносвязная, сверточная, рекуррентная) кодирует *предположения* о структуре данных: *пространство гипотез*, в котором осуществляется поиск хорошей модели. Соответствие выбранной архитектуры данной задаче полностью зависит от соответствия структуры данных предположениям сетевой архитектуры.

Эти разные типы сетей можно объединять для создания больших мультимодальных сетей подобно деталям конструктора LEGO. В некотором смысле слои глубокого обучения — это детали LEGO для обработки информации. Ниже приводится краткий обзор соответствий между некоторыми входными модальностями и сетевыми архитектурами:

- ❑ *Векторные данные* — полносвязные сети (слои `Dense`).
- ❑ *Изображения* — двумерные сверточные сети.
- ❑ *Звуки* (например, *сигналы волновой формы*) — одномерные сверточные сети (предпочтительно) или рекуррентные сети.
- ❑ *Текстовые данные* — одномерные сверточные сети (предпочтительно) или рекуррентные сети.
- ❑ *Временные последовательности* — рекуррентные сети (предпочтительно) или одномерные сверточные сети.
- ❑ *Другие виды последовательных данных* — рекуррентные сети или одномерные сверточные сети. Рекуррентные сети предпочтительнее, если упорядоченность данных имеет большое значение (например, для временных последовательностей, но не для текста).
- ❑ *Видеоданные* — трехмерные сверточные сети (если необходимо захватывать эффекты движения) или комбинация двумерной сверточной сети, действующей на уровне кадров для извлечения признаков, с последующей обработкой рекуррентной сетью или одномерной сверточной сетью для обработки получающихся последовательностей.
- ❑ *Объемные данные* — трехмерные сверточные сети.

Теперь вспомним особенности каждой архитектуры.

## Полносвязные сети

Полносвязные сети — это стек слоев `Dense`, предназначенных для обработки векторных данных (пакетов векторов). Такие сети не предполагают наличия во входных признаках какой-то определенной структуры: они называются *полносвязными* (*densely connected*), потому что измерения слоя `Dense` связаны со всеми другими измерениями. Слой пытается отобразить отношения между любыми двумя входными признаками. Этим он отличается, например, от двумерного сверточного слоя, который рассматривает только *локальные* отношения.

Полносвязные сети чаще всего используются для данных, выражающих качественные характеристики (например, когда входные признаки являются списками атрибутов), таких как данные в наборе с ценами на жилье в Бостоне, который использовался в главе 3. Они также применяются для заключительной классификации или регрессии в большинстве сетей. Например, сверточные сети, рассматривавшиеся

в главе 5, а также рекуррентные сети, рассматривавшиеся в главе 6, обычно завершаются одним или двумя слоями `Dense`.

Помните: для бинарной классификации стек слоев должен завершаться слоем `Dense` с единственным измерением, функцией активации `sigmoid` и функцией потерь `binary_crossentropy`. Вашей целью должно быть значение 0 или 1:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(num_input_
features,)))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

Для выполнения *однозначной классификации* (когда каждый образец принадлежит точно одному классу) завершайте стек слоев слоем `Dense` с количеством измерений, равным количеству классов, и функцией активации `softmax`. Если цели получены прямым кодированием, используйте функцию потерь `categorical_crossentropy`; если они — целые числа, используйте `sparse_categorical_crossentropy`:

```
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(num_input_
features,)))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

Для выполнения *многозначной классификации* (когда каждый образец может принадлежать нескольким классам сразу) завершайте стек слоев слоем `Dense` с количеством измерений, равным количеству классов, функцией активации `softmax` и функцией потерь `binary_crossentropy`. Ваши цели должны быть получены k-мерным прямым кодированием:

```
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(num_input_
features,)))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

Чтобы выполнить регрессию в направлении вектора непрерывных значений, завершайте стек слоев слоем `Dense` с количеством измерений, равным количеству значений, которые вы пытаетесь предсказать (часто одно, например цена на недвижимость), без функции активации. Для регрессии можно использовать несколько

функций потерь; наиболее часто на практике используются `mean_squared_error` (MSE) и `mean_absolute_error` (MAE):

```
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(num_input_
features,)))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_values))

model.compile(optimizer='rmsprop', loss='mse')
```

## Сверточные сети

Сверточные слои выделяют локальные пространственные шаблоны, применяя одни и те же геометрические преобразования к разным участкам в пространстве (*фрагментам*) во входном тензоре. В результате получаются представления, *инвариантные в отношении переноса*, что делает свертки высокоэффективными и модульными. Эта идея применима к пространствам любой размерности: одномерным (последовательностям), двумерным (изображениям), трехмерным (объемам) и т. д. Вы можете использовать слой `Conv1D` для обработки последовательностей (особенно текста — он плохо подходит для обработки временных последовательностей, которые часто не соответствуют предположению о инвариантности в отношении переноса), слой `Conv2D` — для обработки изображений и слой `Conv3D` — для обработки объемов.

*Сверточные нейронные сети* состоят из стека сверточных слоев и слоев выбора максимальных значений по соседям (max-pooling). Слой выбора позволяет снижать пространственную размерность данных, что необходимо для сохранения размеров карты признаков в разумных пределах с ростом числа признаков, и дает возможность последующим сверточным слоям «увидеть» входное пространство на большем протяжении. Сверточные сети часто заканчиваются операцией `Flatten` или слоем глобального выбора, превращающими карту пространственных признаков в векторы, за которыми следуют слои `Dense`, реализующие классификацию или регрессию.

Обратите внимание: высока вероятность того, что в скором времени обычные свертки будут по большей части (или полностью) вытеснены эквивалентной, но более быстрой и эффективной с точки зрения выделения представлений альтернативой: *раздельными свертками по глубине* (depthwise separable convolution, слой `SeparableConv2D`). Это относится к трех-, двух- и одномерным входным данным. При создании новых сетей с нуля использование раздельных сверток по глубине определенно более предпочтительно. Слой `SeparableConv2D` можно использовать как прямую замену слою `Conv2D`, в результате получится более быстрая сеть меньших размеров, лучше справляющаяся с поставленной задачей.

Вот типичная сеть для классификации изображений (в данном случае категориальная классификация):

```
model = models.Sequential()
model.add(layers.SeparableConv2D(32, 3, activation='relu',
                               input_shape=(height, width, channels)))
model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.GlobalAveragePooling2D())

model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

## Рекуррентные нейронные сети

*Рекуррентные нейронные сети* (Recurrent Neural Networks, RNN) обрабатывают входные последовательности по одному временному интервалу за раз, поддерживая *состояние* на всем протяжении (обычно состояние — это вектор или набор векторов: точка в геометрическом пространстве состояний). Обычно они предпочтительнее одномерных сверточных сетей, когда обрабатываются последовательности, где интересующие шаблоны не инвариантны в отношении смещения по времени (например, временные ряды данных, в которых недавнее прошлое важнее отдаленного).

В Keras доступны три слоя RNN: `SimpleRNN`, `GRU` и `LSTM`. Для большинства практических применений лучше использовать `GRU` или `LSTM`. `LSTM` — более мощный из этих двух, но и более затратный в вычислительном смысле; `GRU` можно рассматривать как более простую и незатратную альтернативу.

Чтобы уложить в стек несколько слоев RNN, каждый предыдущий слой перед последним должен возвращать полную последовательность своих выходов (каждый входной временной интервал будет соответствовать выходному); если сверху не накладываются никакие другие слои RNN, тогда сеть возвращает только последний вывод, содержащий информацию обо всей последовательности.

Вот единственный простой слой RNN для бинарной классификации последовательностей векторов:

```
model = models.Sequential()
model.add(layers.LSTM(32, input_shape=(num_timesteps, num_features)))
model.add(layers.Dense(num_classes, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

А вот стек из слоев RNN для бинарной классификации последовательностей векторов:

```
model = models.Sequential()
model.add(layers.LSTM(32, return_sequences=True,
                      input_shape=(num_timesteps, num_features)))
model.add(layers.LSTM(32, return_sequences=True))
model.add(layers.LSTM(32))
model.add(layers.Dense(num_classes, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

### 9.1.7. Пространство возможностей

Что можно построить, использовав приемы глубокого обучения? Помните, что конструирование моделей глубокого обучения напоминает игру с конструктором LEGO: слои можно подключать друг к другу для отображения практически всего, что угодно, при наличии подходящего набора обучающих данных и возможности получения отображения с помощью последовательности геометрических преобразований с разумной сложностью. Пространство возможностей бесконечно. В этом разделе демонстрируется несколько примеров, чтобы показать, что глубокое обучение позволяет решать не только задачи классификации и регрессии, которые традиционно были хлебом наущным для машинного обучения.

Я отсортировал предлагаемые мною примеры применения по модальностям входов и выходов. Обратите внимание на то, что некоторые из них расширяют рамки возможного: хотя можно обучить модель на всех этих задачах, в некоторых случаях такая модель, вероятно, не сможет обеспечить обобщение за границами круга обучающих данных. В разделах 9.2 и 9.3 рассказывается, как эти ограничения могут быть сняты в будущем.

□ Отображение вектора данных в вектор данных:

- *прогнозное здравоохранение* — предсказание результатов лечения по медицинским картам пациентов;
- *анализ поведения* — предсказание продолжительности пребывания пользователя на веб-сайте по множеству атрибутов этого сайта;
- *контроль качества продукции* — предсказание по множеству атрибутов экземпляра произведенного продукта вероятности того, что он перестанет пользоваться спросом в будущем году.

□ Отображение изображения в вектор данных:

- *помощник доктора* — предсказание наличия опухоли по медицинским фотографиям;
- *транспорт с автоматическим управлением* — определение угла поворота рулевых колес по кадрам, поступающим с видеокамеры;

- *настольные игры с ИИ* — предсказание следующего хода игрока по расположению фигур на шахматной доске или камней на доске Го;
  - *помощник диетолога* — предсказание калорийности блюда по его изображению;
  - *предсказание возраста* — определение возраста людей по их автопортретам (селфи).
- Отображение временных последовательностей в вектор данных:
- *прогноз погоды* — прогноз погоды на следующую неделю в определенном местоположении по временным последовательностям метеорологических данных;
  - *интерфейс мозг-компьютер* — отображение временных последовательностей данных магнитной энцефалограммы в команды для компьютера;
  - *анализ поведения* — определение вероятности того, что пользователь купит что-то, по временной последовательности взаимодействий его с веб-сайтом.
- Отображение текста в текст:
- *интеллектуальный автоответчик* — генерирование односторонних ответов на электронные письма;
  - *ответы на вопросы* — генерирование ответов на общие вопросы;
  - *резюмирование* — преобразование длинных статей в краткие обзоры;
- Отображение изображений в текст:
- *генерирование подписей* — генерирование коротких подписей к изображениям, описывающих их содержимое.
- Отображение текста в изображения:
- *генерирование изображений по условию* — получение изображений, соответствующих коротким текстовым описаниям;
  - *выбор/генерирование логотипов* — создание логотипа по названию и краткому описанию компании.
- Отображение изображений в изображения:
- *увеличение разрешения* — отображение изображений с низким разрешением в версии с высоким разрешением;
  - *придание визуальной глубины* — создание карт глубины по плоским изображениям.
- Отображение изображений и текста в текст:
- *вопросы/ответы по изображениям* — отображение изображений и вопросов об их содержимом на естественном языке в ответы на естественном языке.

□ Отображение видео и текста в текст:

- *вопросы/ответы по видео* — отображение видео и вопросов об их содержимом на естественном языке в ответы на естественном языке.

Возможно *почти всё*, но *не совсем всё*. Давайте в следующем разделе посмотрим, чего нельзя сделать с глубоким обучением.

## 9.2. Ограничения глубокого обучения

Пространство возможных применений глубокого обучения почти бесконечно. И все же есть практические области, в которых глубокое обучение оказывается бессильным даже при наличии огромного объема данных, классифицированных человеком. Представьте, например, что у вас есть возможность собрать сотни тысяч — или даже миллионы — описаний функций программного продукта на естественном языке, написанных специалистами, а также соответствующий исходный код, разработанный группой инженеров и реализующий эти функции. Даже с таким объемом вы не сможете обучить модель глубокого обучения читать описание продукта и генерировать соответствующий код. Это лишь один пример из множества. Вообще все, что требует рассуждений, как программирование или применение научных методов долгосрочного планирования и алгоритмического манипулирования данными, недоступно для моделей глубокого обучения, независимо от объема обучающих данных. Даже обучение глубокой нейронной сети простой сортировке — весьма трудоемкая задача.

Это связано с тем, что модель глубокого обучения — всего лишь цепочка *простых геометрических преобразований*, отображающих одно векторное пространство в другое. Она может только отображать одну совокупность данных  $X$  в другую совокупность  $Y$ , предполагая существование обучаемого непрерывного преобразования из  $X$  в  $Y$ . Модель глубокого обучения можно интерпретировать как разновидность программы; но *большинство программ нельзя выразить в виде моделей глубокого обучения* — для большинства задач либо не существует соответствующей глубокой нейронной сети, способной решить ее, либо, если даже она существует, она может быть *необучаемой*: соответствующее геометрическое преобразование может быть чересчур сложным, или могут отсутствовать данные, необходимые для ее обучения.

Масштабирование современных методов глубокого обучения путем увеличения числа слоев и использования больших объемов обучающих данных может лишь слегка смягчить некоторые из этих проблем. Однако это не решает главных проблем, ограничивающих выразительные возможности моделей глубокого обучения, из-за которых большинство программ, которые вы, возможно, захотите включить в обучение, нельзя выразить как последовательность геометрических преобразований совокупности данных.

### 9.2.1. Риск очеловечивания моделей глубокого обучения

На современном этапе развития ИИ существует реальный риск неверно истолковать то, что делают модели глубокого обучения, и переоценить их возможности. Фундаментальной особенностью человека является наша теория разума: наше стремление проецировать намерения, убеждения и знания на окружающий мир. Рисунок улыбающегося лица на скале делает ее «счастливой» — в наших умах. Применительно к глубокому обучению это означает, что, когда, например, нам удается успешно обучить модель, генерирующую подписи к изображениям, мы склонны думать, что модель «понимает» изображенное на них и генерирует подписи. Но потом мы удивляемся, когда любое отступление от изображений, имеющихся в обучающем наборе, заставляет модель генерировать совершенно абсурдные подписи (рис. 9.1).

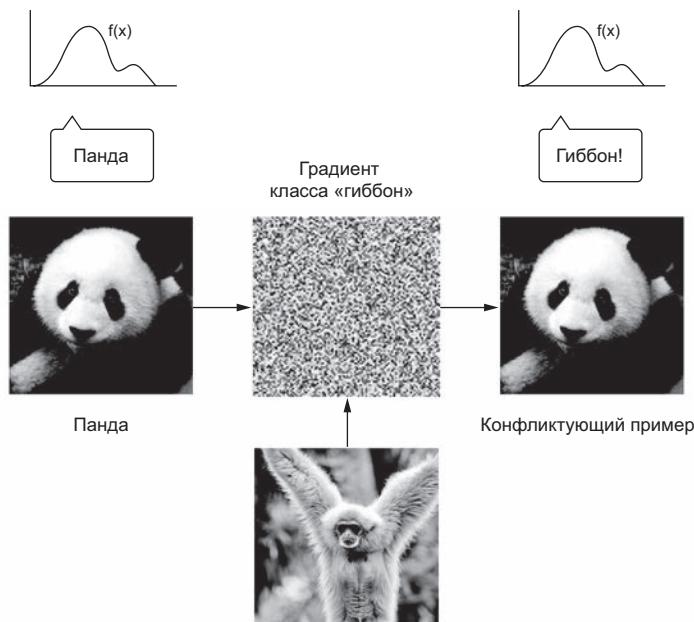


Мальчик, держащий бейсбольную биту.

**Рис. 9.1.** Ошибка системы создания подписей к изображениям, основанной на глубоком обучении

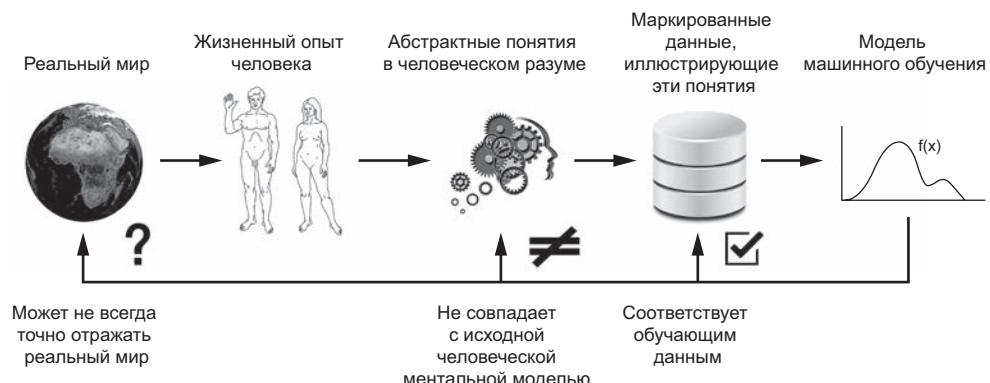
Это особенно ярко подчеркивается примерами с состязательными сетями, которые передают в сеть глубокого обучения образцы, сконструированные специально для того, чтобы обмануть модель. Вы уже знаете, что можно, например, выполнить градиентное восхождение во входном пространстве и сгенерировать входные данные, максимизирующие функцию активации некоторого сверточного фильтра, — это основа приема визуализации фильтров, представленного в главе 5, а также алгоритма DeepDream, который мы обсуждали в главе 8. Аналогично, с помощью градиентного восхождения можно немного изменить изображение, чтобы увеличить вероятность выбора данного класса при классификации. Сделав снимок панды и добавив в него градиент гиббона, можно заставить нейронную сеть классифицировать панду как гиббона (рис. 9.2). Это свидетельство хрупкости таких моделей и является глубоким отличием их отображения входов в выходы от человеческого восприятия.

Проще говоря, модели глубокого обучения не имеют никакого понимания данных, получаемых на входе, — по крайней мере, не в человеческом смысле. Наше собственное понимание изображений, звуков и языка основано на сенсомоторном человеческом опыте. Модели машинного обучения не имеют такого опыта и поэтому



**Рис. 9.2.** Незаметные изменения в изображении могут мешать модели правильно его классифицировать

не могут понимать входные данные подобно человеку. Аннотируя большие количества обучающих примеров для передачи в модели, мы учим их геометрическим преобразованиям, отображающим данные в человеческие понятия на конкретном наборе примеров, но это всего лишь схематический эскиз представлений, имеющихся в наших умах и полученных в результате жизненного опыта. Это подобно тусклому отражению в зеркале (рис. 9.3).



**Рис. 9.3.** Современные модели машинного обучения: подобие отражения в зеркале

Как практик, занимающийся машинным обучением, всегда помните об этом и никогда не попадайте в ловушку, полагая, что нейронные сети понимают решаемую ими задачу — это не так, по крайней мере, не так, как понимаем ее мы. Они обучаются решению другой, намного более узкой задачи, чем нам хотелось бы: отображать обучающие входные данные в целевые данные, точка за точкой. Стоит вам показать им что-то, что отклоняется от обучающих данных, и они начнут проявлять абсурдное поведение.

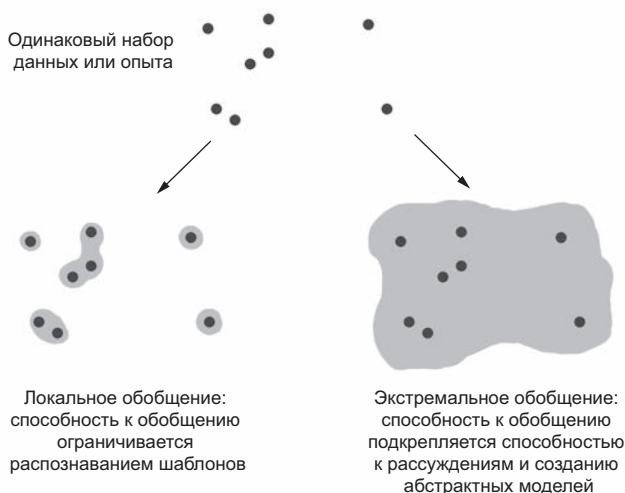
### 9.2.2. Локальное и экстремальное обобщение

Существует фундаментальное отличие простого геометрического преобразования входных данных в выходные, которое выполняют модели глубокого обучения, от того, как думают и обучаются люди. Дело не только в том, что люди учатся на собственном опыте, а не на явных обучающих примерах. В дополнение к разным процессам обучения имеется коренное отличие в природе базовых представлений.

Люди способны на большее, чем просто отображать прямые воздействия в прямые реакции, как глубокая сеть или, может быть, как насекомое. Мы выстраиваем сложные абстрактные модели нашей текущей ситуации, нас самих и других людей и можем использовать эти модели для прогнозирования возможных вариантов развития будущего и долгосрочного планирования. Мы можем соединять известные понятия для представления чего-то, что мы никогда не испытывали прежде: например, изобразить лошадь в джинсах или представить, что мы будем делать, выиграв в лотерее. Эта способность строить гипотезы, расширять пространство нашей ментальной модели за границы того, что мы можем испытывать непосредственно, — *обобщать и рассуждать* — возможно, является определяющей характеристикой человеческого мышления. Я называю это *экстремальным обобщением* (*extreme generalization*): способность адаптироваться к новым, прежде не испытанным ситуациям, имея небольшой объем данных или даже не имея их вообще.

Это резко контрастирует с действиями глубоких сетей, которые я называю *локальным обобщением* (рис. 9.4). Отображение входных данных в выходные, выполняемое глубокой сетью, быстро теряет смысл, если появляются новые входные данные, пусть даже немного отличающиеся от тех, что сеть видела в процессе обучения. Рассмотрим, например, задачу определения параметров запуска ракеты, которая должна сесть на Луну. Если для ее решения использовать глубокую сеть и обучить ее, использовав подход контролируемого обучения или обучения с подкреплением, вам придется накопить результаты тысяч или даже миллионов пробных пусков: вы должны будете передать сети *плотную выборку* из входного пространства, чтобы обучить ее надежно отображать входное пространство в выходное. Мы, будучи людьми, напротив, можем использовать нашу способность к обобщению и придумать физические модели — науку о ракетах — чтобы получить *точное* решение,

которое поможет посадить ракету на Луну после одной или нескольких попыток. Аналогично, если вы решите создать глубокую сеть для управления человеческим телом и пожелаете обучить ее безопасно перемещаться по городу, не попадая под автомобили, вашей сети придется пройти через много тысяч разных гибельных ситуаций, пока она не научится делать вывод о том, что автомобили опасны, и вырабатывает соответствующее поведение уклонения. Однако при попадании в другой город сети придется забыть большую часть того, что она уже изучила, и переучиваться заново. Люди, напротив, способны обучаться правилам безопасного поведения, не переживая фатального конца ни разу, — и снова благодаря своей способности абстрактного моделирования гипотетических ситуаций.



**Рис. 9.4.** Локальное и экстремальное обобщение

Проще говоря, несмотря на наш прогресс в области машинного восприятия, мы все еще очень далеко от ИИ уровня человека. Наши модели способны только к локальному обобщению, адаптируясь к новым ситуациям, которые должны быть похожи на предыдущие, тогда как человеческое сознание способно к долгосрочному планированию и экстремальному обобщению, быстро адаптируясь к радикально отличающимся ситуациям.

### 9.2.3. Подведение итогов

Запомните: единственным реальным успехом глубокого обучения до сих пор была возможность отобразить пространство X в пространство Y с использованием непрерывных геометрических преобразований и больших объемов данных, класси-

фицированных человеком. Решение этой задачи в корне изменило ситуацию во всех отраслях, но мы все еще очень далеки от ИИ уровня человека.

Чтобы преодолеть некоторые ограничения, обсуждавшиеся выше, и создать ИИ, способный полностью заменить мозг человека, мы должны уйти от прямого отображения входных данных в выходные и заняться моделированием способности к рассуждениям и обобщению. Вероятным подходящим субстратом для абстрактного моделирования разных ситуаций и понятий являются компьютерные программы. Мы уже говорили, что модели машинного обучения можно рассматривать как обучаемые программы; в настоящее время мы можем обучить сети только созданию узкого подмножества специальных программ. Однако представьте, что будет, если у нас появится возможность обучить их создавать любые, модульные и многоразовые, программы. Давайте посмотрим в следующем разделе, как этого можно было бы достичь.

## 9.3. Будущее глубокого обучения

Это самый гипотетический раздел, цель которого — расширить горизонты для тех, кто желает присоединиться к существующей исследовательской программе или заняться своими исследованиями. Зная, как действуют глубокие сети, и понимая текущее положение дел в сфере исследований, можем ли мы предсказать направление движения в среднесрочной перспективе? Далее приводятся некоторые мои личные мысли. Имейте в виду, что у меня нет хрустального шара, поэтому многим моим ожиданиям, может быть, не суждено стать реальностью. Я разделяю эти прогнозы не потому, что их состоятельность будет доказана в ближайшем будущем, а потому, что они интересны и выглядят реальными в настоящем.

Вот основные направления, которые мне кажутся многообещающими:

- ❑ *Модели, более близкие к универсальным компьютерным программам*, построенные на основе более широкого набора примитивов, чем современные дифференцируемые слои. Именно так мы приблизимся к возможности моделирования рассуждений и обобщения, отсутствие которой является основным недостатком современных моделей.
- ❑ *Новые формы обучения, делающие возможным предыдущий пункт*, которые позволяют моделям отойти от дифференцируемых преобразований.
- ❑ *Модели, требующие меньше участия людей-инженеров*. Это не ваша задача — бесконечно крутить ручки настройки.
- ❑ *Расширение систематического повторного использования прежде извлеченных признаков и сконструированных архитектур*, с созданием систем метаобучения, использующих модульные подпрограммы.

Обратите внимание: эти соображения не относятся к какому-то конкретному виду контролируемого обучения, которое до сих пор остается хлебом насущным

глубокого обучения, скорее они применимы к любой форме машинного обучения, включая неконтролируемое и самоконтролируемое обучение, а также обучение с подкреплением. Принципиально не важно, откуда берутся размеченные данные или как выглядит цикл обучения; это всего лишь разные ветви машинного обучения — разные грани одной и той же конструкции. Давайте рассмотрим их поближе.

### 9.3.1. Модели как программы

Как отмечалось в предыдущем разделе, одна из обязательных трансформаций в сфере машинного обучения, которые мы можем ожидать, — это уход от моделей, реализующих лишь *распознавание шаблонов* и способных только на *локальные обобщения*, в сторону моделей, способных *абстрагировать* и *рассуждать* и тем самым достигать *экстремального обобщения*. Все современные программы ИИ, способные на простейшие рассуждения, написаны человеком-программистом: например, программы, опирающиеся на алгоритмы поиска, манипулирование графами и формальную логику. В игре AlphaGo компании DeepMind, например, большая часть интеллекта спроектирована и запрограммирована опытными программистами с применением четких алгоритмов (таких, как алгоритм Монте-Карло для поиска в деревьях); обучение на данных происходит только в специализированных модулях (оценочные и стратегические сети). Однако в будущем такие системы ИИ могут стать полностью обучаемыми без участия человека.

Как такое может произойти? Рассмотрим хорошо изученный тип сетей: рекуррентные нейронные сети (RNN). Важно отметить, что RNN имеют немного меньше ограничений, чем сети прямого распространения, потому что RNN — это чуть больше, чем простые геометрические преобразования: это геометрические преобразования, *многократно повторяемые во внутреннем цикле for*. Сам временной цикл `for` «зашит» человеком-разработчиком: это предположение, имплантированное в сеть. Естественно, рекуррентные сети все еще очень ограничены в возможности представления, в первую очередь потому, что каждый их шаг является дифференцируемым геометрическим преобразованием, и они переносят информацию из шага в шаг через точки в непрерывном геометрическом пространстве (векторы состояний). Теперь вообразите нейронную сеть, дополненную программными примитивами, но вместо единственного жестко зашитого цикла `for` с четко определенной геометрической памятью она включает в себя обширный набор программных примитивов, которыми может свободно манипулировать и расширять свои функции обработки, организуя ветвление с помощью инструкции `if`, выполняя условные циклы `while`, создавая переменные, используя диск в качестве долговременного хранилища, применяя операции сортировки, используя сложные структуры данных (например, списки, графы и хеш-таблицы) и многое другое. Пространство программ, которые такая сеть сможет представить, было бы намного шире, чем то, которое можно представить с помощью современных моделей глубокого обучения, и некоторые из этих программ могли бы достигать высочайшей степени обобщения.

Мы одновременно уйдем от жестко запрограммированного интеллекта (программного обеспечения, написанного вручную) и от обучаемого геометрического интеллекта (глубокое обучение). Вместо этого мы получим сочетание формальных алгоритмических, поддерживающих возможность абстрагирования и рассуждения модулей и геометрических модулей, поддерживающих неформальное знание и распознавание шаблонов. Вся система будет обучаться без участия или с минимальным участием человека.

Родственная подобласть ИИ, которая, как мне кажется, может «взлететь», — это *синтез программ*, в частности синтез нейронных программ. Синтез программ заключается в создании простых программ с использованием алгоритма поиска (возможно, генетического поиска, как в генетическом программировании) для исследования обширного пространства возможных программ. Поиск останавливается при обнаружении программы, соответствующей заданным требованиям, часто имеющим форму множества пар ввод/вывод. Это очень напоминает машинное обучение: по заданным обучающим данным, имеющим форму пар ввод/вывод, мы находим программу, которая соответствует входным и выходным данным и способна обобщать новые входные данные. Различие в том, что вместо обучения значений параметров в четко определенной программе (нейронной сети) мы генерируем исходный код посредством процесса дискретного поиска.

Я определенно ожидаю увидеть в этой области новую волну интереса в ближайшие несколько лет. В частности, я ожидаю появления новой смежной области между глубоким обучением и синтезом программ, где вместо программ на языке общего назначения будут генерироваться нейронные сети (потоки геометрической обработки данных), дополненные широким набором алгоритмических примитивов, таких как циклы `for`, и многих других (рис. 9.5). Это должно быть более практично и полезно, чем прямой синтез исходного кода, и существенно расширит диапазон задач, поддающихся решению с применением машинного обучения, — пространство программ, которые можно автоматически генерировать на основе обучающих данных. Современные рекуррентные сети можно считать предтечами таких гибридных алгоритмически-геометрических моделей.



**Рис. 9.5.** Программа, сгенерированная одновременно на основе геометрических (распознавание шаблонов, предсказание) и алгоритмических (рассуждения, поиск, память) примитивов

### 9.3.2. За границами алгоритма обратного распространения ошибки и дифференцируемых слоев

Если модели машинного обучения станут похожими на программы, они перестанут быть дифференцируемыми — эти программы по-прежнему будут использовать непрерывные дифференцируемые геометрические слои в качестве подпрограмм, но сами модели потеряют это качество. Как результат, станет невозможна использование алгоритм обратного распространения ошибки для корректировки весовых значений сети в процессе обучения моделей — по крайней мере, он перестанет быть единственным. Нам потребуется эффективный способ обучения недифференцируемых систем. Среди современных подходов можно назвать генетические алгоритмы, эволюционные стратегии, некоторые виды обучения с подкреплением и метод чередующихся направлений множителя (Alternating Direction Method of Multipliers, ADMM). Естественно, градиентный спуск никуда не денется; градиентная информация всегда будет полезна для оптимизации дифференцируемых параметрических функций. Однако наши модели станут более сложными, нежели простые дифференцируемые параметрические функции, и, следовательно, их автоматическое развитие (*обучение в машинном обучении*) потребует больше, чем может дать простой алгоритм обратного распространения ошибки.

Алгоритм обратного распространения ошибки прекрасно подходит для выявления хороших цепочечных преобразований, но он неэффективен с точки зрения расходования вычислительных ресурсов, потому что не в полной мере использует преимущества модульной организации глубоких сетей. Есть один универсальный рецепт повышения эффективности: ввести модульность и иерархию. То есть алгоритм обратного распространения ошибки можно сделать более эффективным, внедрив раздельное обучение модулей с механизмом синхронизации между ними и организовав некоторое подобие иерархии. Эта стратегия нашла отражение в недавней работе компании DeepMind, связанной с синтетическими градиентами. В ближайшем будущем я ожидаю больших успехов в этом направлении. Я могу представить будущее, когда в целом недифференцируемые модели (но состоящие из дифференцируемых элементов) будут обучаться — выращиваться — с использованием эффективного процесса поиска, без использования градиентов, а дифференцируемые части будут обучаться еще быстрее за счет применения более эффективной версии алгоритма обратного распространения ошибки.

### 9.3.3. Автоматизированное машинное обучение

В будущем архитектуры моделей будут формироваться в ходе обучения, а не определяться вручную инженерами. Автоматическое формирование архитектур немыслимо без использования широких наборов примитивов и моделей машинного обучения, похожих на программы.

В настоящее время основная работа инженера, занимающегося глубоким обучением, состоит из обработки данных с помощью сценариев на Python и тщательной настройки архитектуры и гиперпараметров глубокой сети для получения рабочей модели — или даже суперсовременной модели, если инженер достаточно честолюбив. Нет нужды говорить о том, что это не оптимальный подход. Но ИИ может помочь в этом. К сожалению, этап подготовки данных трудно автоматизировать, потому что для этого инженер часто должен знать предметную область и четко понимать на высоком уровне, чего он пытается добиться. Однако настройка гиперпараметров — это простая процедура поиска и в этом случае известно, чего добивается инженер: это определяется функцией потерь настраиваемой сети. В настоящее время уже принято использовать простые системы *AutoML*, которые способны взять на себя самый тяжелый труд настройки моделей. Я даже создал свою такую систему несколько лет тому назад, чтобы победить в состязании на Kaggle.

На самом простом уровне такая система могла бы настраивать множество слоев в стеке, их порядок и количество размерностей или фильтров в каждом слое. Обычно это делается с использованием библиотек, таких как Hyperopt, которая обсуждалась в главе 7. Но можно пойти дальше и попытаться получить подходящую архитектуру с нуля, с минимально возможными ограничениями: например, используя обучение с подкреплением или генетические алгоритмы.

Другое важное направление применения AutoML — обучение архитектур моделей с весами. Обучение новой модели с нуля всякий раз, когда мы пробуем немного другой архитектуры, крайне неэффективно, поэтому по-настоящему мощная система AutoML могла бы развивать архитектуры одновременно с настройкой признаков модели через обратное распространение ошибки на обучающих данных. Такие решения уже начинают появляться, когда я пишу эти строки.

Когда это начнет происходить, инженеры глубокого обучения не лишатся работы — вместо этого они займутся цепочками создания ценностей. Они будут прикладывать больше усилий для разработки сложных функций потерь, точнее отражающих бизнес-цели, и изучения влияния их моделей на цифровую экосистему, в которой они развертываются (например, на пользователей, потребляющих прогнозы моделей и генерирующих обучающие данные), что в настоящее время могут позволить себе только крупные компании.

### 9.3.4. Непрерывное обучение и повторное использование модульных подпрограмм

Когда модели станут сложнее и будут основаны на более насыщенных алгоритмических примитивах, эта повышенная сложность потребует увеличить степень повторного использования результатов прежних решений вместо обучения новых моделей с нуля каждый раз, когда возникает новая задача или новый набор данных. Многие наборы данных содержат недостаточно информации, чтобы мы могли приступить к созданию новых, сложных моделей с нуля, и поэтому необходимо будет

использовать информацию из прежних наборов данных (представьте, что вам пришлось бы изучать русский язык с нуля всякий раз, когда вы открываете новую книгу, — это было бы просто невозможно). Обучение моделей с нуля для каждой новой задачи неэффективно также из-за большого перекрытия между текущими задачами и прежними.

В последние годы неоднократно отмечалось интересное наблюдение: обучение *одной и той же* модели для решения мало связанных между собой задач дает в результате модель, которая *лучше подходит для каждой задачи*. Например, обучение одной и той же нейронной модели машинного перевода с английского на немецкий и с французского на итальянский дает в результате модель, которая лучше подходит для каждой пары языков. Аналогично, одновременное обучение модели классификации и сегментации изображений с использованием одной и той же сверточной основы дает в результате модель, которая лучше решает обе задачи. Это вполне объяснимо: в малосвязанных задачах всегда *какая-то часть* информации является общей, в результате объединенная модель получает доступ к большему объему информации о каждой отдельной задаче, нежели модель, обучаемая для решения какой-то конкретной задачи.

В настоящее время под повторным использованием моделей в разных задачах подразумевается использование обученных весов моделей, выполняющих универсальные функции, такие как выделение визуальных признаков. Пример этого вы видели в главе 5. В будущем я ожидаю, что в обход войдет более обобщенная версия: мы будем использовать не только ранее извлеченные признаки (веса подмоделей), но также архитектуры моделей и процедуры обучения. По мере того как модели будут становиться все более похожими на программы, мы начнем повторно использовать подпрограммы подобно классам и функциям в обычных языках программирования.

Представьте современный процесс разработки программного обеспечения: решив определенную задачу (например, поддержку HTTP-запросов в Python), инженер тут же упаковывает решение в абстрактную библиотеку многократного пользования. Другие инженеры, столкнувшись с подобной проблемой в будущем, смогут отыскать существующие библиотеки, загрузить их и использовать в своих проектах. Похожим способом в будущем системы метаобучения смогут собирать новые программы, просеивая глобальную библиотеку высокогенеративных блоков многократного пользования. Когда система обнаружит, что ей нужны схожие подпрограммы для нескольких разных задач, она сможет создать абстрактную многоразовую версию подпрограммы и сохранить ее в глобальной библиотеке (рис. 9.6). Такой процесс реализует *абстракцию*: необходимый компонент для достижения экстремального обобщения. О подпрограмме, полезной для решения различных задач в разных областях, можно сказать, что она *абстрактная* в отношении некоторых аспектов решаемой задачи. Это определение абстракции похоже на понятие абстракции в разработке программного обеспечения. Такие подпрограммы могут быть геометрическими (модули глубокого обучения с предварительно выделенными представлениями) или алгоритмическими (ближе к библиотекам, которыми пользуются современные программисты).



**Рис. 9.6.** Система метаобучения, способная быстро разрабатывать модели для конкретных задач, используя примитивы многократного пользования (алгоритмические и геометрические), и таким способом достигать экстремального обобщения

### 9.3.5. Долгосрочная перспектива

Вот какой я вижу долгосрочную перспективу машинного обучения:

- ❑ Модели будут больше похожи на программы и будут обладать возможностями, выходящими далеко за рамки непрерывных геометрических преобразований входных данных, которые мы используем в настоящее время. Эти программы, вероятно, будут ближе к абстрактным ментальным моделям, которые люди выстраивают в своем сознании, и будут способны к более широкому обобщению благодаря богатой алгоритмической природе.
- ❑ Модели будут сочетать в себе алгоритмические модули, реализующие возможность формальных рассуждений, поиск и средства абстрагирования с геометрическими модулями, обеспечивающими неформальное знание и распознавание шаблонов. AlphaGo (система, для создания которой потребовалось программное обеспечение, созданное вручную, и множество решений, принятых людьми) является собой ранний пример того, как может выглядеть такое сочетание символьического и геометрического ИИ.
- ❑ Такие модели будут создаваться автоматически, без участия людей-инженеров, с использованием модульных компонентов, хранящихся в глобальной библиотеке подпрограмм многократного пользования — библиотеке, накапливающей высококачественные модели, обученные ранее на тысячах задач и наборов данных. Часто встречающиеся шаблоны решений задач будут идентифициро-

ваться системой метаобучения, превращаться в подпрограммы многоократного пользования — подобно функциям и классам в разработке программного обеспечения — и добавляться в глобальную библиотеку. Это приведет к *абстракции*.

- Эта глобальная библиотека и связанная с ней система моделей смогут достичь уровня экстремального обобщения, сопоставимого с человеческим: для новой задачи или ситуации система сможет сконструировать новую работающую модель, использовав очень небольшой объем данных, благодаря широте программных примитивов, поддерживающих обобщение, и обширному опыту решения похожих задач. Точно так же люди быстро осваивают новую сложную видеоигру, опираясь на прежний опыт использования других видеоигр, а не основываясь на простом отображении стимулов в действия. Так происходит потому, что модели, сформированные на базе предыдущего опыта, являются абстрактными и похожими на программы.
- Такую непрерывно развивающуюся систему моделей можно рассматривать как *общий искусственный интеллект* (Artificial General Intelligence, AGI). Однако не нужно ожидать, что в результате возникнет какой-то необычный апокалиптический робот: это чистая фантазия, порожденная длинной последовательностью глубоких недоразумений и непонимания как интеллекта, так и технологий. Впрочем, такая критика не является целью данной книги.

## 9.4. Как не отстать от прогресса в быстроразвивающейся области

На прощание я хочу дать вам несколько советов, как продолжать учиться и расширять свои знания и навыки после того, как вы перевернете последнюю страницу этой книги. Современному глубокому обучению, каким мы его знаем, всего несколько лет, несмотря на долгую предысторию, уходящую корнями в прошлое на несколько десятилетий. Благодаря экспоненциальному росту финансовых вливаний и числа исследователей начиная с 2013 года, в настоящее время эта область развивается очень интенсивно. Знания, полученные в этой книге, не останутся актуальными навсегда, кроме того, здесь рассказывалось далеко не обо всем, что может вам пригодиться в вашей карьере.

К счастью, в интернете существует множество бесплатных ресурсов, с помощью которых вы сможете оставаться в курсе текущего положения дел и расширять свои горизонты. Вот некоторые из них.

### 9.4.1. Практические решения реальных задач на сайте Kaggle

Один из самых эффективных способов приобрести практический опыт — поучаствовать в состязаниях по машинному обучению на сайте Kaggle (<https://kaggle.com>).

сом). Единственный действенный способ научиться что-то делать — практика и фактическое программирование, вот в чем состоит философия этой книги. А состязания на сайте Kaggle — это естественное ее продолжение. На Kaggle вы найдете массу постоянно обновляющихся заданий, многие из которых связаны с глубоким обучением. Эти задания подготовлены компаниями, заинтересованными в получении новых решений некоторых из наиболее сложных проблем машинного обучения. Победителям предлагаются довольно внушительные призы.

Большинство состязаний было выиграно с использованием библиотеки XGBoost (поверхностное машинное обучение) или фреймворка Keras (глубокое обучение). Таким образом, вы вполне подготовлены к участию! Поучаствовав в нескольких состязаниях, возможно, в составе команды, вы познакомитесь с практической стороной некоторых передовых приемов, описанных в этой книге: настройкой гиперпараметров, преодолением проблемы переобучения на проверочном наборе данных и ансамблированием моделей.

#### 9.4.2. Знакомство с последними разработками на сайте arXiv

Исследования в области глубокого обучения, в отличие от других направлений в науке, полностью открыты. Публикуемые статьи доступны всем желающим, как и масса сопутствующего программного кода, распространяемого с открытым исходным кодом. arXiv (<https://arxiv.org>) — произносится как «архив» (под буквой Х в данном случае подразумевается греческая буква «хи» — χ) — это открытый препринт-сервер для размещения статей в области физики, математики и информатики. Он фактически стал основным средоточием ультрасовременных знаний о машинном и глубоком обучении. Подавляющее большинство исследователей глубокого обучения выгружают на сайт arXiv свои статьи, написанные вскоре после состязаний. Это позволяет им поднять флаг и заявить о конкретных находках, не дожидаясь решения конференции (для чего могут потребоваться месяцы), что абсолютно необходимо, учитывая быстрые темпы исследований и высокую конкуренцию в этой области. Это также поддерживает чрезвычайно высокий темп развития области: все новые находки немедленно становятся доступными для всех желающих.

Существенной проблемой является ежедневное появление в arXiv большого количества новых статей, что делает невозможным хотя бы бегло ознакомиться с ними со всеми, а тот факт, что они не подвергаются экспертной оценке, усложняет выявление наиболее важных и ценных из них. С каждым днем становится все труднее выделить полезный сигнал из шума. В настоящее время эта проблема не имеет хорошего решения. Но есть некоторые инструменты, которые могут оказать хоть какую-то помощь: вспомогательный сайт arXiv Sanity Preserver (<http://arxiv-sanity.com>) играет роль рекомендательного механизма при выборе новых статей и может помочь вам следить за новыми разработками в определенном узком сегменте глубокого обучения. Также можно использовать Google Scholar (<https://scholar.google.com>), чтобы отслеживать выход новых публикаций определенных авторов.

### 9.4.3. Исследование экосистемы Keras

По состоянию на ноябрь 2017 года насчитывалось примерно 200 000 пользователей, и их число продолжает расти. Вокруг фреймворка Keras сложилась огромная экосистема из руководств, справочников и проектов с открытым исходным кодом:

- ❑ Основной справочник по фреймворку Keras — электронная документация, доступная по адресу: <https://keras.io>. Исходный код Keras можно найти по адресу: <https://github.com/fchollet/keras>.
- ❑ Задавать вопросы, получать помощь и принимать участие в обсуждении проблем глубокого обучения можно на канале Keras в Slack: <https://kerasteam.slack.com>.
- ❑ В блоге Keras (<https://blog.keras.io>) вы найдете руководства по Keras и другие статьи, связанные с глубоким обучением.
- ❑ Вы можете следить за мной в Twitter: @fchollet.

## 9.5. Заключительное слово

Вот и закончилась книга «Глубокое обучение с Python»! Надеюсь, вы узнали кое-что новое о машинном обучении, глубоком обучении, Keras и, может быть, даже о способности мыслить в целом. Обучение — это пожизненное путешествие, особенно в области ИИ, где неизвестностей гораздо больше, чем определенности. Поэтому продолжайте учиться, задавайте вопросы и занимайтесь исследованиями. Никогда не останавливайтесь. Потому что, даже несмотря на достигнутый прогресс, многие фундаментальные вопросы в ИИ пока не имеют ответа. А многие вопросы даже еще не были правильно сформулированы.

# ПРИЛОЖЕНИЕ А

## Установка Keras и его зависимостей в Ubuntu

Процедура подготовки рабочей станции для экспериментов в области глубокого обучения довольно сложна и включает в себя следующие шаги, которые мы подробно рассмотрим в этом приложении:

1. Установка пакетов научных вычислений для Python — Numpy и SciPy — и библиотеки подпрограмм линейной алгебры (Basic Linear Algebra Subprogram, BLAS) в целях увеличения скорости работы моделей на CPU.
2. Установка двух дополнительных пакетов, которые могут пригодиться при работе с Keras: HDF5 (для сохранения больших нейронных сетей в файлы) и Graphviz (для визуализации архитектур нейронных сетей).
3. Добавление поддержки выполнения кода глубокого обучения на GPU путем установки драйверов CUDA и cuDNN.
4. Установка низкоуровневой библиотеки для Keras: TensorFlow, CNTK или Theano.
5. Установка Keras.

Процедура может показаться удручающе сложной. Однако в действительности самое сложное в ней — это настройка поддержки GPU. Если отказаться от нее, весь процесс можно выполнить всего несколькими командами за пару минут.

Допустим, что у вас имеется свежеустановленная ОС Ubuntu на компьютере, оснащенном NVIDIA GPU. Прежде чем начать, убедитесь, что у вас установлен диспетчер пакетов для Python `pip` и обновлен кэш пакетов `apt`:

```
$ sudo apt-get update  
$ sudo apt-get upgrade  
$ sudo apt-get install python-pip python-dev
```

## PYTHON 2 И PYTHON 3

По умолчанию для установки пакетов Python, таких как `pythonpip`, в Ubuntu используется Python 2. Если вы решите использовать Python 3, используйте префикс `python3` вместо `python`. Например:

```
$ sudo apt-get install python3-pip python3-dev
```

Когда будете устанавливать пакеты с помощью диспетчера `pip`, имейте в виду, что по умолчанию он нацелен на Python 2. Чтобы перенацелить его на Python 3, используйте `pip3`:

```
$ sudo pip3 install tensorflow-gpu
```

# A.1. Установка пакетов научных вычислений для Python

Если вы пользуетесь Mac, рекомендуем устанавливать пакеты научных вычислений для Python с помощью сборника Anaconda ([www.continuum.io/downloads](http://www.continuum.io/downloads)). Обратите внимание на то, что в этот сборник не входят HDF5 и Graphviz, которые вы должны установить вручную. Выполните следующие шаги, чтобы *вручную* установить пакеты научных вычислений для Python в Ubuntu:

1. Установите библиотеку BLAS (в данном случае OpenBLAS), чтобы получить поддержку быстрых операций с тензорами на CPU:

```
$ sudo apt-get install build-essential cmake git unzip \
  pkg-config libopenblas-dev liblapack-dev
```

2. Установите пакеты научных вычислений для Python: Numpy, SciPy и Matplotlib. Они необходимы для выполнения любых вычислений, связанных с машинным или глубоким обучением в Python:

```
$ sudo apt-get install python-numpy python-scipy python- matplotlib
  python-yaml
```

3. Установите HDF5. Эта библиотека, первоначально разработанная в NASA, позволяет сохранять огромные объемы числовых данных в файлах в эффективном двоичном формате. Она позволит вам быстро и эффективно сохранять свои модели Keras на диске:

```
$ sudo apt-get install libhdf5-serial-dev python-h5py
```

4. Установите Graphviz и pydot-ng, два пакета, которые помогут вам визуализировать модели Keras. Они не нужны фреймворку Keras для работы, поэтому

вы можете пропустить этот шаг и вернуться к нему, когда вам потребуются эти пакеты. Вот команды, которые устанавливают пакеты:

```
$ sudo apt-get install graphviz  
$ sudo pip install pydot-ng
```

5. Установите дополнительные пакеты, использующиеся в некоторых наших примерах кода:

```
$ sudo apt-get install python-opencv
```

## A.2. Настройка поддержки GPU

Поддержка GPU не является строго необходимой, но крайне желательна. Код всех примеров, что приводятся в этой книге, можно выполнить на CPU ноутбука, но тогда в некоторых случаях вам придется ждать по несколько часов, пока модель обучится, вместо нескольких минут на хорошем GPU. Если у вас нет современного NVIDIA GPU, можете пропустить этот шаг и сразу перейти к разделу А.3.

Чтобы использовать свой NVIDIA GPU в глубоком обучении, вы должны установить два пакета:

- ❑ *CUDA* – набор драйверов поддержки GPU, позволяющих запускать на нем низкоуровневый программный код, осуществляющий параллельные вычисления;
- ❑ *cuDNN* – библиотека оптимизированных примитивов для глубокого обучения. При использовании cuDNN и выполнении на GPU скорость обучения моделей обычно можно поднять от 50 до 100%.

Пакет TensorFlow зависит от конкретных версий драйверов CUDA и библиотеки cuDNN. На момент написания этих строк этот пакет использовал CUDA версии 8 и cuDNN версии 6. Более подробную информацию о совместимости версий вы найдете на сайте TensorFlow: [www.tensorflow.org/install/install\\_linux](http://www.tensorflow.org/install/install_linux).

Для установки выполните следующие шаги:

1. Загрузите пакет CUDA. Для Ubuntu (и других разновидностей Linux) NVIDIA собирает готовый к использованию пакет, доступный для загрузки по адресу: <https://developer.nvidia.com/cuda-downloads>:

```
$ wget http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/  
→x86_64/cuda-repo-ubuntu1604_9.0.176-1_amd64.deb
```

2. Установите пакет CUDA. В Ubuntu проще всего установку выполнить с помощью диспетчера пакетов *apt*. В этом случае вы легко сможете устанавливать обновления с помощью *apt*, как только они будут доступны:

```
$ sudo dpkg -i cuda-repo-ubuntu1604_9.0.176-1_amd64.deb  
$ sudo apt-key adv --fetch-keys  
→http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/  
→x86_64/7fa2af80.pub
```

```
$ sudo apt-get update  
$ sudo apt-get install cuda-8-0
```

### 3. Установите библиотеку cuDNN:

- Создайте бесплатную учетную запись разработчика NVIDIA (увы, это необходимо, чтобы получить доступ к библиотеке cuDNN для загрузки) и загрузите cuDNN по адресу: <https://developer.NVIDIA.com/cudnn> (выберите версию cuDNN, совместимую с TensorFlow). Как и в случае с CUDA, NVIDIA предоставляет пакеты для разных разновидностей Linux flavors — мы используем версию для Ubuntu 16.04. Обратите внимание: если вы работаете с экземпляром EC2, вам не нужно загружать архив cuDNN непосредственно на этот экземпляр; вместо этого загрузите его на свой локальный компьютер и затем выгрузите в свой экземпляр EC2 (с помощью команды `scp`).

- Установите cuDNN:

```
$ sudo dpkg -i dpkg -i libcudnn6*.deb
```

### 4. Установите TensorFlow:

- TensorFlow, с поддержкой GPU или без нее, можно установить из каталога пакетов PyPI с помощью диспетчера `pip`. Вот команда для установки без поддержки GPU:

```
$ sudo pip install tensorflow
```

- А это команда для установки TensorFlow с поддержкой GPU:

```
$ sudo pip install tensorflow-gpu
```

## A.3. Установка Theano (необязательно)

После установки TensorFlow можно не устанавливать библиотеки Theano поддержки выполнения кода Keras. Но иногда бывает полезно перейти с использования TensorFlow на Theano при строительстве моделей Keras.

Библиотеку Theano также можно установить из PyPI:

```
$ sudo pip install theano
```

Если у вас установлена поддержка GPU, вы должны настроить Theano на его использование. Для этого создайте конфигурационный файл Theano следующей командой:

```
nano ~/.theanorc
```

Затем добавьте в него следующие строки:

```
[global]  
floatX = float32
```

```
device = gpu0
[nvcc]
fastmath = True
```

## A.4. Установка Keras

Установить Keras можно из PyPI:

```
$ sudo pip install keras
```

Также можно установить Keras из репозитория на GitHub. В этом случае вы получаете доступ к папке *keras/examples*, содержащей большое количество примеров сценариев:

```
$ git clone https://github.com/fchollet/keras
$ cd keras
$ sudo python setup.py install
```

После установки можете попробовать запустить сценарий Keras, такой, как следующий пример MNIST:

```
python examples/mnist_cnn.py
```

Обратите внимание на то, что для выполнения этого примера потребуется несколько минут, поэтому можете просто нажать комбинацию Ctrl-C после того, как станет понятно, что сценарий запустился и выполняется нормально.

После хотя бы одного запуска Keras появится его конфигурационный файл *~/.keras/keras.json*. Вы можете открыть его в редакторе и выбрать, какую низкоуровневую библиотеку использовать: **tensorflow**, **theano** или **cntk**. Содержимое конфигурационного файла должно выглядеть так:

```
{
  "image_data_format": "channels_last",
  "epsilon": 1e-07,
  "floatx": "float32",
  "backend": "tensorflow"
}
```

Пока выполняется сценарий *examples/mnist\_cnn.py*, можете проверить нагрузку на GPU в другом окне с командной оболочкой:

```
$ watch -n 5 nvidia-smi -a --display=utilization
```

Вот и всё! Примите поздравления — теперь вы можете начинать создавать свои приложения глубокого обучения.

# ПРИЛОЖЕНИЕ В

## Запуск Jupyter Notebook на экземпляре EC2 GPU

В этом приложении приводится пошаговое руководство по запуску блокнотов (также встречается термин «тетрадь» или «ноутбук») Jupyter Notebook с примерами глубокого обучения на экземпляре AWS GPU и их редактированию из любого места с помощью браузера. Это превосходная среда для исследований в области глубокого обучения, если у вас нет GPU на вашем локальном компьютере. Оригинальную (и самую свежую) версию этого руководства можно найти по адресу: <https://blog.keras.io>.

### B.1. Что такое Jupyter Notebook? Зачем запускать Jupyter Notebook на AWS GPU?

*Jupyter Notebook* – это веб-приложение, позволяющее писать и снабжать комментариями код на Python в интерактивном режиме. Это отличный способ поэкспериментировать, провести исследования и поделиться своими результатами с другими.

Многие приложения глубокого обучения потребляют значительный объем вычислительных ресурсов и могут выполняться часами и даже сутками, особенно на CPU ноутбука. Наличие поддержки GPU может значительно увеличить скорость обучения и получения результатов (часто в 5–10 раз, при переходе с современного CPU на единственный современный GPU). Но не у всех есть GPU на локальной машине. Запуск Jupyter Notebook на AWS дает вам тот же опыт, что и при запуске на локальном компьютере, позволяя при этом использовать один или несколько GPU в экземпляре AWS. И вам придется заплатить только за использованное процессорное время, что может оказаться выгоднее, чем вкладывать деньги в приобретение собственного GPU, особенно если вы занимаетесь глубоким обучением от случая к случаю.

## B.2. Когда нежелательно использовать Jupyter на AWS для глубокого обучения?

Стоимость аренды экземпляров AWS GPU может расти очень быстро. Мы предлагаем использовать экземпляры со стоимостью 0,90 доллара США в час. Они прекрасно подходят для редкого и непродолжительного использования; но если вы собираетесь экспериментировать по нескольку часов в день, тогда вам лучше собрать свой компьютер для глубокого обучения с видеокартой TITAN X или GTX 1080 Ti.

В общем случае используйте Jupyter на EC2, если у вас нет своего GPU или если не хотите устанавливать Keras с зависимостями, например драйверами GPU. Если у вас есть свой GPU, рекомендуем запускать модели локально. В этом случае воспользуйтесь руководством по установке в приложении А.

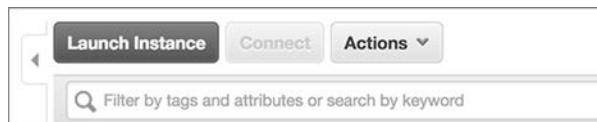
### ПРИМЕЧАНИЕ

Вам понадобится активная учетная запись AWS. Некоторое знакомство с AWS EC2 также не будет лишним, хотя этого и не требуется.

## B.3. Настройка экземпляра AWS GPU

Описываемый далее процесс настройки займет 5–10 минут:

1. Откройте панель управления EC2 (<https://console.aws.amazon.com/ec2/v2>) и щелкните на кнопке **Launch Instance** (Запустить экземпляра) (рис. B.1).

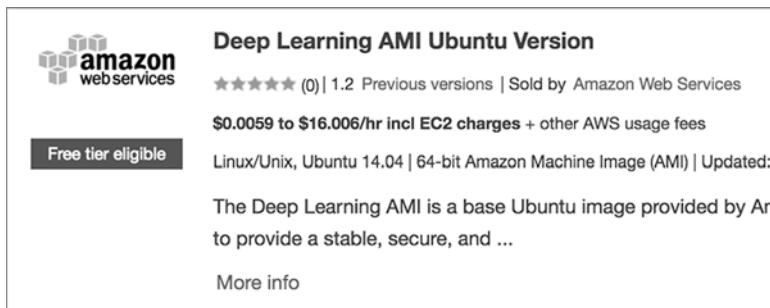


**Рис. B.1.** Панель управления EC2

2. Выберите пункт **AWS Marketplace** (Площадка AWS) (рис. B.2) и введите «deep learning» (глубокое обучение) в поле поиска. Прокрутите страницу вниз и найдите образ виртуальной машины с названием **Deep Learning AMI Ubuntu Version** (Образ машины Amazon для глубокого обучения, версия с Ubuntu) (рис. B.3); выберите его.
3. Выберите экземпляр **p2.xlarge** (рис. B.4). Экземпляры этого типа предоставляют доступ к одному GPU и стоят 0,90 доллара США в час (по состоянию на март 2017-го).



**Рис. В.2.** Выбор площадки EC2 AMI Marketplace



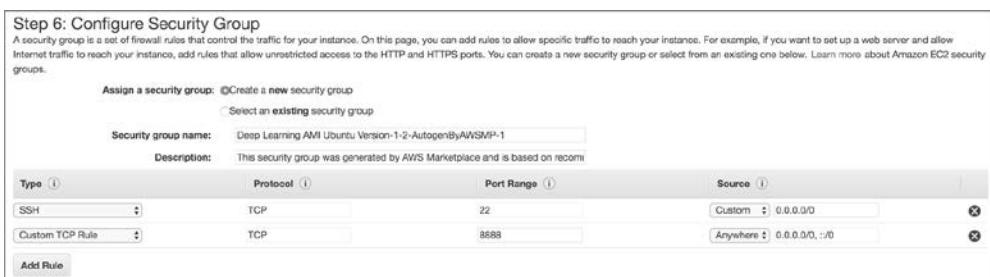
**Рис. В.3.** Образ EC2 для глубокого обучения

1. Choose AMI	2. Choose Instance Type	3. Configure Instance	4. Add Storage
<b>Step 2: Choose an Instance Type</b>			
<input type="checkbox"/>	GPU instances	g2.8xlarge	32
<input checked="" type="checkbox"/>	GPU compute	p2.xlarge	4
<input type="checkbox"/>	GPU compute	p2.8xlarge	32

**Рис. В.4.** Экземпляр p2.xlarge

- Вы можете сохранить конфигурацию по умолчанию на шагах **Configure Instance** (Настройка экземпляра), **Add Storage** (Добавить хранилище) и **Add Tags** (Добавить теги), но обязательно измените настройки на шаге **Configure Security Group** (Настройка группы безопасности). Создайте свое TCP-правило, открывающее порт 8888 (рис. В.5): это правило может разрешить доступ с вашего текущего IP-адреса (например, вашего ноутбука) или любого другого (такого, как 0.0.0.0/0), если первое невозможно. Обратите внимание: если открыть порт 8888 для

доступа с любого IP-адреса, тогда буквально каждый сможет прослушивать этот порт на вашем экземпляре (где вы будете запускать сценарии IPython). Добавьте защиту сценариев паролем, чтобы уменьшить риск их изменения случайным посетителем, — правда, это очень слабая защита. Если это вообще возможно, подумайте об ограничении доступа к конкретному IP-адресу. Но если ваш IP-адрес постоянно изменяется, тогда этот вариант вам не подойдет. Если вы собираетесь открыть доступ с любого IP-адреса, не оставляйте на экземпляре конфиденциальных данных.



**Рис. В.5.** Настройка новой группы безопасности

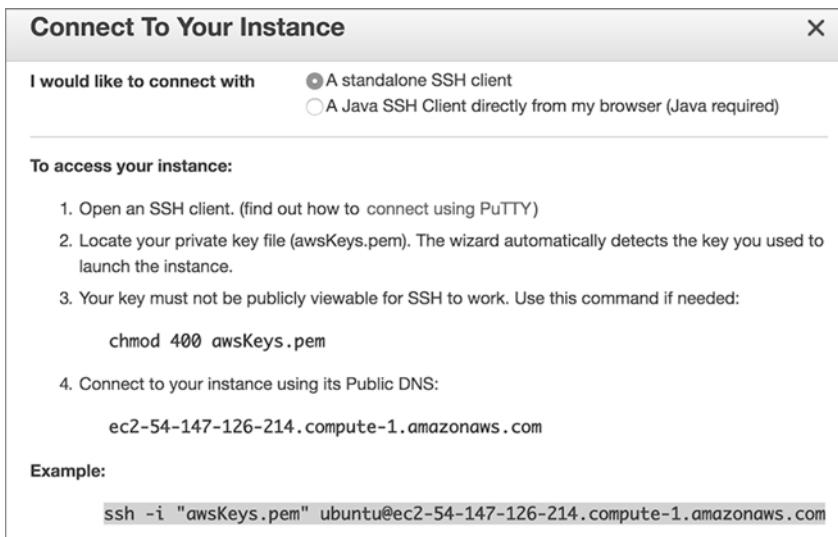
### ПРИМЕЧАНИЕ

По окончании процедуры настройки вам будет предложено сделать выбор между использованием существующих ключей соединения и созданием новых. Если прежде вы не пользовались EC2, создайте новые ключи и загрузите их к себе.

- Чтобы подключиться к своему экземпляру, выберите его в панели управления EC2, щелкните на кнопке **Connect** (Подключиться) и следуйте инструкциям (рис. В.6). Обратите внимание на то, что для загрузки экземпляра может потребоваться несколько минут. Если вам не удастся подключиться с первой попытки, подождите немного и повторите попытку.
- После регистрации в экземпляре через SSH создайте каталог *ssl* в корне файловой системы экземпляра и перейдите в него командой *cd* (это необязательно, но так будет проще выполнять последующие команды):
 

```
$ mkdir ssl
$ cd ssl
```
- Создайте новый сертификат SSL с помощью OpenSSL и файлы *cert.key* и *cert.pem* в текущем каталоге *ssl*:
 

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout "cert.key" -out
->"cert.pem" -batch
```



**Рис. B.6.** Инструкции по подключению

### B.3.1. Настройка Jupyter

Перед использованием Jupyter нужно изменить его конфигурацию по умолчанию. Для этого выполните следующие шаги:

1. Создайте новый конфигурационный файл для Jupyter (выполнив следующую команду на удаленном экземпляре):

```
$ jupyter notebook --generate-config
```

2. Дополнительно можно сгенерировать пароль для доступа к блокнотам Jupyter. Так как кто-то из вас может настроить доступ к своему экземпляру с любого IP-адреса в зависимости от выбора, сделанного при настройке группы безопасности), лучше ограничить доступ к блокнотам Jupyter паролем. Чтобы сгенерировать пароль, откройте оболочку IPython (командой `ipython`) и выполните следующие команды:

```
from IPython.lib import passwd
passwd()
exit
```

3. Команда `passwd()` предложит ввести пароль дважды и затем выведет хеш пароля. Скопируйте хеш — он вскоре понадобится. Хеш выглядит примерно так:

```
sha1:b592a9cf2ec6:b99edb2fd3d0727e336185a0b0eab561aa533a43
```

Обратите внимание: здесь показан хеш для слова *password*, вы должны использовать другой пароль.

- Откройте конфигурационный файл Jupyter с помощью `vi` (или любого другого текстового редактора):

```
$ vi ~/.jupyter/jupyter_notebook_config.py
```

- Этот файл содержит код на Python, в котором все строки закомментированы. Вставьте в начало файла следующие строки:

<b>Путь к закрытому ключу для сгенерированного вами сертификата</b>  <b>Путь к сгенерированному вами сертификату</b>	<b>Встраивание изображения в текст при использовании Matplotlib</b>
<pre>c = get_config() ← Получение объекта с конфигурацией → c.NotebookApp.certfile = u'/home/ubuntu/ssl/cert.pem' → c.NotebookApp.keyfile = u'/home/ubuntu/ssl/cert.key' c.IPKernelApp.pylab = 'inline' ← Обслуживание блокнотов локально c.NotebookApp.ip = '*' ← Обслуживание блокнотов локально</pre>	
<pre>→ c.NotebookApp.open_browser = False c.NotebookApp.password = → 'sha1:b592a9cf2ec6:b99edb2fd3d0727e336185a0b0eab561aa533a43'</pre>	
<b>Не открывать окно браузера по умолчанию, когда используются блокноты Jupyter</b>	<b>Хеш пароля, сгенерированный выше</b>

#### ПРИМЕЧАНИЕ

Если у вас нет опыта использования редактора `vi`, запомните, что вы должны нажать клавишу `I`, чтобы получить возможность вставлять новые символы в текст. Завершив редактирование, нажмите клавишу `Esc`, введите команду `:wq` и нажмите `Enter`, чтобы закрыть редактор `vi` и сохранить изменения (`:wq` означает «write-quit» — «записать-завершить»).

## B.4. Установка Keras

Теперь почти все готово к использованию Jupyter. Осталось только обновить Keras. В образ виртуальной машины Amazon уже установлен фреймворк Keras, но его версия может оказаться устаревшей. Выполните следующую команду в удаленном экземпляре:

```
$ sudo pip install keras --upgrade
```

Если вы используете Python 3 (блокноты Jupyter, сопровождающие эту книгу, используют Python 3), обновите Keras также с помощью pip3:

```
$ sudo pip3 install keras --upgrade
```

Если в экземпляре имеется готовый конфигурационный файл Keras (когда я пишу эти строки, конфигурационный файл по умолчанию не создавался в AMI, но с тех пор многое могло измениться), его следует удалить — на всякий случай. Keras повторно создаст стандартный конфигурационный файл в первой попытке запуска.

Если следующая команда вернет сообщение, что файл не найден, просто проигнорируйте его:

```
$ rm -f ~/.keras/keras.json
```

## B.5. Настройка перенаправления локальных портов

В командной оболочке *на локальной машине* (не в удаленном экземпляре) запустите перенаправление локального порта 443 (порт HTTPS) в порт 8888 удаленного экземпляра:

```
$ sudo ssh -i awsKeys.pem -L локальный_порт:локальный_компьютер:удаленный_порт  
удаленный_компьютер
```

В моем случае полная команда после подстановки всех параметров приобрела такой вид:

```
$ sudo ssh -i awsKeys.pem -L  
→ 443:127.0.0.1:8888 ubuntu@ec2-54-147-126-214.compute-1.amazonaws.com
```

## B.6. Доступ к Jupyter из браузера на локальном компьютере

На удаленном экземпляре клонируйте репозиторий GitHub, содержащий блокноты Jupyter Notebook для этой книги:

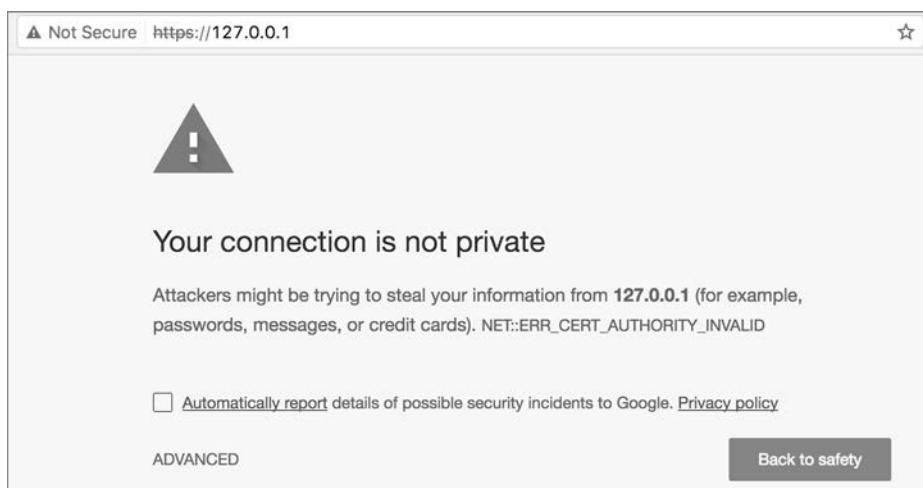
```
$ git clone https://github.com/fchollet/deep-learning-with-python-notebooks.git  
cd deep-learning-with-python-notebooks
```

Запустите Jupyter Notebook следующей командой, выполнив ее на удаленном экземпляре:

```
$ jupyter notebook
```

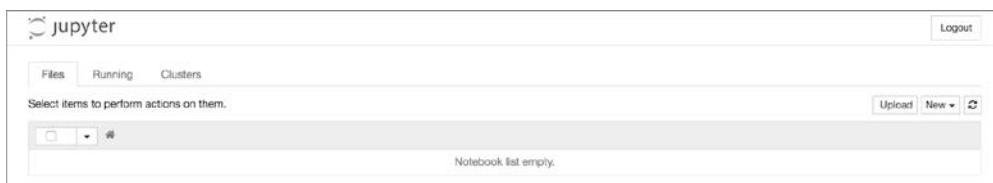
Затем запустите браузер на локальном компьютере и откройте страницу с локальным адресом (<https://127.0.0.1>) — благодаря настроенному перенаправлению портов запрос будет отправлен удаленному процессу Jupyter Notebook. Убедитесь, что указали протокол HTTPS в адресной строке, иначе вы получите ошибку SSL.

В браузере должно появиться предупреждение безопасности, как показано на рис. В.7. Это обусловлено тем, что сгенерированный вами сертификат SSL не проходит проверку в доверенном центре сертификации (в конце концов, вы сами сгенеририровали его). Щелкните на кнопке Advanced (Дополнительно) и продолжите навигацию.

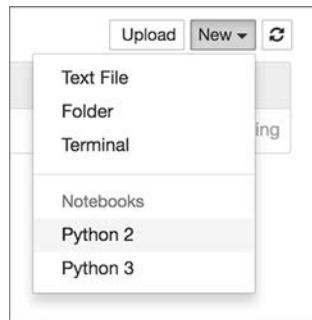


**Рис. В.7.** Предупреждение о безопасности; его можно игнорировать

Далее вам будет предложено ввести пароль Jupyter. После этого перед вами откроется панель управления Jupyter (рис. В.8).



**Рис. В.8.** Панель управления Jupyter



**Рис. В.9.** Создание нового блокнота Jupiter Notebook

Выберите New > Notebook (Новый > Блокнот), чтобы начать работу (рис. В.9). Вы можете выбрать любую из предложенных версий Python. Вот и всё!

Книга Франсуа Шолле "Глубокое обучение на Python" была куплена" в реcкладчину за 10 рублей на сайте опен хайд биз

*Франсуа Шолле*  
**Глубокое обучение на Python**

*Перевел с английского А. Киселев*

*Серия «Библиотека программиста»*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>Е. Самородских</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>Н. Викторова, И. Тимофеева</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург, ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.

Дата изготовления: 04.2018. Назначение: 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 19.04.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 32,250.  
Тираж 1700. Заказ 0000.

# КНИГА-ПОЧТОЙ



**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»  
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- на нашем сайте: [www.piter.com](http://www.piter.com)
- по электронной почте: [books@piter.com](mailto:books@piter.com)
- по телефону: **(812) 703-73-74**

**ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:**

- Ⓐ Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- Ⓑ С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Ⓒ Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- Ⓓ В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

**ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:**

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщают по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте [www.piter.com](http://www.piter.com)).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте [www.piter.com](http://www.piter.com)).

**ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:**

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

- БЕСПЛАТНАЯ ДОСТАВКА:**
- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
  - почтой России при предварительной оплате заказа на сумму **от 2000 руб.**



**ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает  
профессиональную, популярную и детскую развивающую литературу**

**Заказать книги оптом можно в наших представительствах  
РОССИЯ**

**Санкт-Петербург:** м. «Выборгская», Б. Сампсониевский пр., д. 29а  
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

**Москва:** м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж  
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

**Воронеж:** тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

**Екатеринбург:** ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;  
e-mail: office@ekat.piter.com; skype: ekat.manager2

**Нижний Новгород:** тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

**Ростов-на-Дону:** ул. Ульяновская, д. 26  
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

**Самара:** ул. Молодогвардейская, д. 33а, офис 223  
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,  
pitvolga@samara-ttk.ru

**БЕЛАРУСЬ**

**Минск:** ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;  
e-mail: og@minsk.piter.com

**Издательский дом «Питер» приглашает к сотрудничеству авторов:**  
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com  
**Подробная информация здесь:** <http://www.piter.com/page/avtoru>

**Издательский дом «Питер» приглашает к сотрудничеству зарубежных  
торговых партнеров или посредников, имеющих выход на зарубежный  
рынок:** тел./факс: (812) 703-73-73; e-mail: sales@piter.com

---

**Заказ книг для вузов и библиотек:**  
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

**Заказ книг по почте:** на сайте [www.piter.com](http://www.piter.com); тел.: (812) 703-73-74, доб. 6216;  
e-mail: books@piter.com

---

**Вопросы по продаже электронных книг:** тел.: (812) 703-73-74, доб. 6217;  
e-mail: kuznetsov@piter.com