

# Глубокое обучение на Python

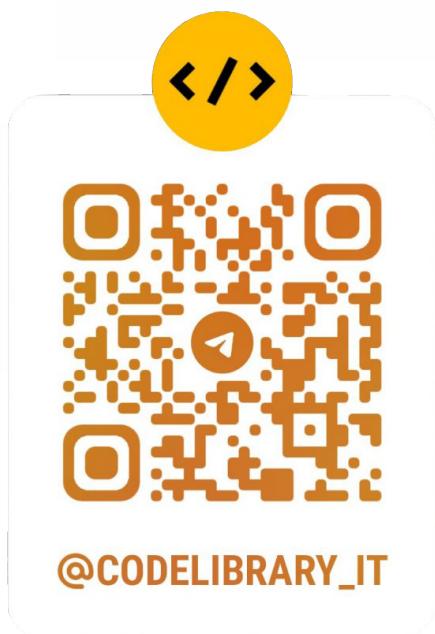
Второе международное  
издание

Франсуа Шолле



# *Deep Learning with Python*

SECOND EDITION



FRANÇOIS CHOLLET



MANNING  
SHELTER ISLAND

# Глубокое обучение на Python

Второе международное издание

Франсуа Шолле



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1

УДК 004.43

Ш78

## Шолле Франсуа

Ш78 Глубокое обучение на Python. 2-е межд. издание. — СПб.: Питер, 2023. — 576 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1909-7

Глубокое обучение динамично развивается, открывая все новые и новые возможности создания ПО. Это не только автоматический перевод текстов с одного языка на другой, распознавание изображений, но и многое другое. Глубокое обучение превратилось в важный навык, необходимый каждому разработчику. Keras и TensorFlow облегчают жизнь разработчикам и позволяют легко разбогатить даже тем, кто не имеет фундаментальных знаний в области математики или науки о данных.

Настала пора познакомиться с глубоким обучением и мощной библиотекой Keras!

В этом расширенном и дополненном издании создатель библиотеки Keras — Франсуа Шолле — делится знаниями и с новичками, и с опытными специалистами. Иллюстрации и наглядные примеры помогут вам разобраться с самыми сложными вопросами и концепциями. Вы быстро приобретете навыки, необходимые для разработки приложений глубокого обучения.

**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 Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

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 978-1617296864 англ.

© by Manning Publications Co. All rights reserved.

ISBN 978-5-4461-1909-7

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2022

© Серия «Библиотека программиста», 2022

# *Краткое содержание*

---

Предисловие . . . . .	18
Благодарности . . . . .	20
О книге . . . . .	21
Об авторе . . . . .	23
Иллюстрация на обложке . . . . .	24
От издательства . . . . .	25
<b>Глава 1.</b> Что такое глубокое обучение . . . . .	26
<b>Глава 2.</b> Математические основы нейронных сетей . . . . .	56
<b>Глава 3.</b> Введение в Keras и TensorFlow . . . . .	103
<b>Глава 4.</b> Начало работы с нейронными сетями: классификация и регрессия . . . . .	135
<b>Глава 5.</b> Основы машинного обучения . . . . .	165
<b>Глава 6.</b> Обобщенный процесс машинного обучения . . . . .	203
<b>Глава 7.</b> Работа с Keras: глубокое погружение . . . . .	226
<b>Глава 8.</b> Введение в глубокое обучение в технологиях компьютерного зрения . . . . .	260
<b>Глава 9.</b> Продвинутые приемы глубокого обучения в технологиях компьютерного зрения . . . . .	303
<b>Глава 10.</b> Глубокое обучение на временных последовательностях . . . . .	350
<b>Глава 11.</b> Глубокое обучение для текста . . . . .	384
<b>Глава 12.</b> Генеративное глубокое обучение . . . . .	449
<b>Глава 13.</b> Методы и приемы для применения на практике . . . . .	506
<b>Глава 14.</b> Заключение . . . . .	529

# *Оглавление*

---

Предисловие . . . . .	18
Благодарности . . . . .	20
О книге. . . . .	21
Кому адресована эта книга . . . . .	21
О примерах кода . . . . .	22
Об авторе . . . . .	23
Иллюстрация на обложке . . . . .	24
От издательства . . . . .	25
Глава 1. Что такое глубокое обучение. . . . .	26
1.1. Искусственный интеллект, машинное и глубокое обучение . . . . .	27
1.1.1. Искусственный интеллект . . . . .	27
1.1.2. Машинное обучение . . . . .	28
1.1.3. Изучение правил и представлений данных . . . . .	30
1.1.4. «Глубина» глубокого обучения . . . . .	33
1.1.5. Принцип действия глубокого обучения в трех картинках . . . . .	35
1.1.6. Какой ступени развития достигло глубокое обучение . . . . .	37
1.1.7. Не верьте рекламе . . . . .	38
1.1.8. Перспективы ИИ . . . . .	39
1.2. Что было до глубокого обучения: краткая история машинного обучения . . . . .	40
1.2.1. Вероятностное моделирование . . . . .	40
1.2.2. Первые нейронные сети. . . . .	41

1.2.3. Ядерные методы . . . . .	41
1.2.4. Деревья решений, случайные леса и градиентный бустинг . . . . .	43
1.2.5. Назад к нейронным сетям . . . . .	44
1.2.6. Отличительные черты глубокого обучения . . . . .	45
1.2.7. Современный ландшафт машинного обучения . . . . .	46
1.3. Почему глубокое обучение? Почему сейчас? . . . . .	49
1.3.1. Оборудование . . . . .	49
1.3.2. Данные . . . . .	51
1.3.3. Алгоритмы . . . . .	51
1.3.4. Новая волна инвестиций . . . . .	52
1.3.5. Демократизация глубокого обучения . . . . .	54
1.3.6. Ждать ли продолжения этой тенденции? . . . . .	54
<b>Глава 2. Математические основы нейронных сетей</b> . . . . .	56
2.1. Первое знакомство с нейронной сетью . . . . .	57
2.2. Представление данных для нейронных сетей . . . . .	61
2.2.1. Скаляры (тензоры нулевого ранга) . . . . .	61
2.2.2. Векторы (тензоры первого ранга) . . . . .	62
2.2.3. Матрицы (тензоры второго ранга) . . . . .	62
2.2.4. Тензоры третьего и более высоких рангов . . . . .	62
2.2.5. Ключевые атрибуты . . . . .	63
2.2.6. Манипулирование тензорами с помощью NumPy . . . . .	64
2.2.7. Пакеты данных . . . . .	65
2.2.8. Практические примеры тензоров с данными . . . . .	66
2.2.9. Векторные данные . . . . .	66
2.2.10. Временные ряды или последовательности . . . . .	67
2.2.11. Изображения . . . . .	67
2.2.12. Видео . . . . .	68
2.3. Шестеренки нейронных сетей: операции с тензорами . . . . .	69
2.3.1. Поэлементные операции . . . . .	70
2.3.2. Расширение . . . . .	71
2.3.3. Скалярное произведение тензоров . . . . .	73
2.3.4. Изменение формы тензора . . . . .	75
2.3.5. Геометрическая интерпретация операций с тензорами . . . . .	76
2.3.6. Геометрическая интерпретация глубокого обучения . . . . .	80

2.4. Механизм нейронных сетей: оптимизация на основе градиента . . . . .	81
2.4.1. Что такое производная . . . . .	83
2.4.2. Производная операций с тензорами: градиент. . . . .	84
2.4.3. Стохастический градиентный спуск . . . . .	86
2.4.4. Объединение производных: алгоритм обратного распространения ошибки . . . . .	89
2.5. Оглядываясь на первый пример . . . . .	95
2.5.1. Повторная реализация первого примера в TensorFlow . . . . .	97
2.5.2. Выполнение одного этапа обучения . . . . .	99
2.5.3. Полный цикл обучения . . . . .	100
2.5.4. Оценка модели . . . . .	101
Краткие итоги главы . . . . .	101
<b>Глава 3.</b> Введение в Keras и TensorFlow . . . . .	103
3.1. Что такое TensorFlow . . . . .	104
3.2. Что такое Keras . . . . .	105
3.3. Keras и TensorFlow: краткая история . . . . .	106
3.4. Настройка окружения для глубокого обучения. . . . .	107
3.4.1. Jupyter Notebook: предпочтительный способ проведения экспериментов с глубоким обучением . . . . .	109
3.4.2. Использование Colaboratory. . . . .	109
3.5. Первые шаги с TensorFlow . . . . .	112
3.5.1. Тензоры-константы и тензоры-переменные . . . . .	113
3.5.2. Операции с тензорами: математические действия в TensorFlow . . . . .	115
3.5.3. Второй взгляд на GradientTape . . . . .	116
3.5.4. Полный пример: линейный классификатор на TensorFlow. . . . .	117
3.6. Анатомия нейронной сети: знакомство с основами Keras. . . . .	122
3.6.1. Слои: строительные блоки глубокого обучения. . . . .	122
3.6.2. От слоев к моделям . . . . .	126
3.6.3. Этап «компиляции»: настройка процесса обучения . . . . .	128
3.6.4. Выбор функции потерь . . . . .	130
3.6.5. Метод fit() . . . . .	130
3.6.6. Оценка потерь и метрик на проверочных данных. . . . .	131
3.6.7. Вывод: использование модели после обучения . . . . .	133
Краткие итоги главы . . . . .	134

---

<b>Глава 4.</b> Начало работы с нейронными сетями: классификация и регрессия . . . . .	135
4.1. Классификация отзывов к фильмам: пример бинарной классификации . . . . .	137
4.1.1. Набор данных IMDB . . . . .	137
4.1.2. Подготовка данных . . . . .	139
4.1.3. Конструирование модели . . . . .	140
4.1.4. Проверка решения . . . . .	143
4.1.5. Использование обученной сети для предсказаний на новых данных . . . . .	146
4.1.6. Дальнейшие эксперименты . . . . .	147
4.1.7. Подведение итогов . . . . .	147
4.2. Классификация новостных лент: пример классификации в несколько классов . . . . .	148
4.2.1. Набор данных Reuters . . . . .	148
4.2.2. Подготовка данных . . . . .	149
4.2.3. Конструирование модели . . . . .	150
4.2.4. Проверка решения . . . . .	151
4.2.5. Предсказания на новых данных . . . . .	153
4.2.6. Другой способ обработки меток и потерь . . . . .	154
4.2.7. Важность использования достаточно больших промежуточных слоев . . . . .	154
4.2.8. Дальнейшие эксперименты . . . . .	155
4.2.9. Подведение итогов . . . . .	155
4.3. Предсказание цен на дома: пример регрессии . . . . .	156
4.3.1. Набор данных с ценами на жилье в Бостоне . . . . .	156
4.3.2. Подготовка данных . . . . .	157
4.3.3. Конструирование модели . . . . .	158
4.3.4. Оценка решения методом перекрестной проверки по К блокам . . . . .	159
4.3.5. Предсказания на новых данных . . . . .	163
4.3.6. Подведение итогов . . . . .	163
Краткие итоги главы . . . . .	163
<b>Глава 5.</b> Основы машинного обучения . . . . .	165
5.1. Обобщение: цель машинного обучения . . . . .	165
5.1.1. Недобучение и переобучение . . . . .	166
5.1.2. Природа общности в глубоком обучении . . . . .	172

5.2. Оценка моделей машинного обучения . . . . .	180
5.2.1. Обучающие, проверочные и контрольные наборы данных . . . . .	180
5.2.2. Выбор базового уровня . . . . .	184
5.2.3. Что важно помнить об оценке моделей . . . . .	185
5.3. Улучшение качества обучения модели . . . . .	185
5.3.1. Настройка основных параметров градиентного спуска . . . . .	186
5.3.2. Использование более удачной архитектуры . . . . .	187
5.3.3. Увеличение емкости модели . . . . .	188
5.4. Улучшение общности . . . . .	190
5.4.1. Курирование набора данных . . . . .	190
5.4.2. Конструирование признаков . . . . .	191
5.4.3. Ранняя остановка . . . . .	193
5.4.4. Регуляризация модели . . . . .	193
Краткие итоги главы . . . . .	202
<b>Глава 6. Обобщенный процесс машинного обучения . . . . .</b>	<b>203</b>
6.1. Определение задачи . . . . .	205
6.1.1. Формулировка задачи . . . . .	205
6.1.2. Сбор данных . . . . .	207
6.1.3. Первичный анализ данных . . . . .	211
6.1.4. Выбор меры успеха . . . . .	212
6.2. Разработка модели . . . . .	212
6.2.1. Подготовка данных . . . . .	213
6.2.2. Выбор протокола оценки . . . . .	215
6.2.3. Преодоление базового случая . . . . .	215
6.2.4. Следующий шаг: разработка модели с переобучением . . . . .	217
6.2.5. Регуляризация и настройка модели . . . . .	218
6.3. Развёртывание модели . . . . .	219
6.3.1. Объяснение особенностей работы модели заинтересованным сторонам и обозначение границ ожидаемого . . . . .	219
6.3.2. Предоставление доступа к модели . . . . .	220
6.3.3. Мониторинг качества работы модели в процессе эксплуатации . . . . .	223
6.3.4. Обслуживание модели . . . . .	224
Краткие итоги главы . . . . .	225

---

<b>Глава 7.</b> Работа с Keras: глубокое погружение . . . . .	226
7.1. Спектр рабочих процессов . . . . .	227
7.2. Разные способы создания моделей Keras . . . . .	227
7.2.1. Последовательная модель Sequential . . . . .	228
7.2.2. Функциональный API . . . . .	231
7.2.3. Создание производных от класса Model . . . . .	239
7.2.4. Смешивание и согласование различных компонентов . . . . .	241
7.2.5. Используйте правильный инструмент . . . . .	242
7.3. Встроенные циклы обучения и оценки . . . . .	243
7.3.1. Использование собственных метрик . . . . .	244
7.3.2. Использование обратных вызовов . . . . .	245
7.3.3. Разработка своего обратного вызова . . . . .	247
7.3.4. Мониторинг и визуализация с помощью TensorBoard . . . . .	249
7.4. Разработка своего цикла обучения и оценки . . . . .	251
7.4.1. Обучение и прогнозирование . . . . .	252
7.4.2. Низкоуровневое использование метрик . . . . .	253
7.4.3. Полный цикл обучения и оценки . . . . .	254
7.4.4. Ускорение вычислений с помощью tf.function . . . . .	256
7.4.5. Использование fit() с нестандартным циклом обучения . . . . .	257
Краткие итоги главы . . . . .	259
<b>Глава 8.</b> Введение в глубокое обучение в технологиях компьютерного зрения . . . . .	260
8.1. Введение в сверточные нейронные сети . . . . .	261
8.1.1. Операция свертывания . . . . .	264
8.1.2. Выбор максимального значения из соседних (max-pooling) . . . . .	269
8.2. Обучение сверточной нейронной сети с нуля на небольшом наборе данных . . . . .	272
8.2.1. Целесообразность глубокого обучения для решения задач с небольшими наборами данных . . . . .	272
8.2.2. Загрузка данных . . . . .	273
8.2.3. Конструирование сети . . . . .	276
8.2.4. Предварительная обработка данных . . . . .	278
8.2.5. Обогащение данных . . . . .	283

8.3. Использование предварительно обученной модели . . . . .	288
8.3.1. Выделение признаков . . . . .	289
8.3.2. Дообучение предварительно обученной модели . . . . .	298
Краткие итоги главы . . . . .	302
<b>Глава 9.</b> Продвинутые приемы глубокого обучения в технологиях компьютерного зрения . . . . .	303
9.1. Три основные задачи в сфере компьютерного зрения . . . . .	303
9.2. Пример сегментации изображения . . . . .	305
9.3. Современные архитектурные шаблоны сверточных сетей . . . . .	313
9.3.1. Модульность, иерархия, многократное использование . . . . .	314
9.3.2. Остаточные связи . . . . .	317
9.3.3. Пакетная нормализация . . . . .	321
9.3.4. Раздельная свертка по глубине . . . . .	324
9.3.5. Собираем все вместе: мини-модель с архитектурой Xception . . . . .	326
9.4. Интерпретация знаний, заключенных в сверточной нейронной сети . . . . .	329
9.4.1. Визуализация промежуточных активаций . . . . .	330
9.4.2. Визуализация фильтров сверточных нейронных сетей . . . . .	337
9.4.3. Визуализация тепловых карт активации класса . . . . .	343
Краткие итоги главы . . . . .	349
<b>Глава 10.</b> Глубокое обучение на временных последовательностях . . . . .	350
10.1. Разные виды временных последовательностей . . . . .	350
10.2. Пример прогнозирования температуры . . . . .	352
10.2.1. Подготовка данных . . . . .	355
10.2.2. Базовое решение без привлечения машинного обучения . . . . .	359
10.2.3. Базовое решение с привлечением машинного обучения . . . . .	360
10.2.4. Попытка использовать одномерную сверточную модель . . . . .	362
10.2.5. Первое базовое рекуррентное решение . . . . .	364
10.3. Рекуррентные нейронные сети . . . . .	366
10.3.1. Рекуррентный слой в Keras . . . . .	369

---

10.4. Улучшенные методы использования рекуррентных нейронных сетей . . . . .	373
10.4.1. Использование рекуррентного прореживания для борьбы с переобучением . . . . .	374
10.4.2. Наложение нескольких рекуррентных слоев друг на друга . . . . .	377
10.4.3. Использование двунаправленных рекуррентных нейронных сетей . . . . .	379
10.4.4. Что дальше . . . . .	382
Краткие итоги главы . . . . .	383
<b>Глава 11.</b> Глубокое обучение для текста . . . . .	384
11.1. Обработка естественных языков . . . . .	384
11.2. Подготовка текстовых данных . . . . .	387
11.2.1. Стандартизация текста . . . . .	388
11.2.2. Деление текста на единицы (токенизация) . . . . .	389
11.2.3. Индексирование словаря . . . . .	390
11.2.4. Использование слоя TextVectorization . . . . .	392
11.3. Два подхода к представлению групп слов: множества и последовательности . . . . .	396
11.3.1. Подготовка данных IMDB с отзывами к фильмам . . . . .	397
11.3.2. Обработка наборов данных: мешки слов . . . . .	399
11.3.3. Обработка слов как последовательностей: модели последовательностей . . . . .	406
11.4. Архитектура Transformer . . . . .	417
11.4.1. Идея внутреннего внимания . . . . .	417
11.4.2. Многоголовое внимание . . . . .	423
11.4.3. Кодировщик Transformer . . . . .	424
11.4.4. Когда использовать модели последовательностей вместо моделей мешка слов . . . . .	431
11.5. За границами классификации текста: обучение «последовательность в последовательность» . . . . .	432
11.5.1. Пример машинного перевода . . . . .	434
11.5.2. Обучение типа «последовательность в последовательность» рекуррентной сети . . . . .	437
11.5.3. Обучение типа «последовательность в последовательность» архитектуры Transformer . . . . .	442
Краткие итоги главы . . . . .	448

<b>Глава 12.</b> Генеративное глубокое обучение . . . . .	449
12.1. Генерирование текста . . . . .	451
12.1.1. Краткая история генеративного глубокого обучения для генерирования последовательностей . . . . .	451
12.1.2. Как генерируются последовательности данных . . . . .	452
12.1.3. Важность стратегии выбора . . . . .	453
12.1.4. Реализация генерации текста в Keras . . . . .	456
12.1.5. Обратный вызов для генерации текста с разными значениями температуры . . . . .	460
12.1.6. Подведение итогов . . . . .	463
12.2. DeepDream . . . . .	464
12.2.1. Реализация DeepDream в Keras . . . . .	465
12.2.2. Подведение итогов . . . . .	472
12.3. Нейронная передача стиля . . . . .	473
12.3.1. Функция потерь содержимого . . . . .	474
12.3.2. Функция потерь стиля . . . . .	474
12.3.3. Нейронная передача стиля в Keras . . . . .	475
12.3.4. Подведение итогов . . . . .	481
12.4. Генерирование изображений с вариационными автокодировщиками . . . . .	482
12.4.1. Выбор шаблонов из скрытых пространств изображений . .	482
12.4.2. Концептуальные векторы для редактирования изображений . . . . .	483
12.4.3. Вариационные автокодировщики . . . . .	484
12.4.4. Реализация VAE в Keras . . . . .	487
12.4.5. Подведение итогов . . . . .	493
12.5. Введение в генеративно-состязательные сети . . . . .	493
12.5.1. Реализация простейшей генеративно-состязательной сети . . . . .	495
12.5.2. Набор хитростей . . . . .	496
12.5.3. Получение набора данных CelebA . . . . .	497
12.5.4. Дискриминатор . . . . .	498
12.5.5. Генератор . . . . .	499
12.5.6. Состязательная сеть . . . . .	501
12.5.7. Подведение итогов . . . . .	504
Краткие итоги главы . . . . .	505

---

<b>Глава 13.</b> Методы и приемы для применения на практике . . . . .	506
13.1. Получение максимальной отдачи от моделей . . . . .	507
13.1.1. Оптимизация гиперпараметров . . . . .	507
13.1.2. Ансамблирование моделей . . . . .	515
13.2. Масштабирование обучения моделей . . . . .	517
13.2.1. Ускорение обучения на GPU со смешанной точностью . . . . .	518
13.2.2. Обучение на нескольких GPU . . . . .	522
13.2.3. Обучение на TPU . . . . .	525
Краткие итоги главы . . . . .	528
<b>Глава 14.</b> Заключение . . . . .	529
14.1. Краткий обзор ключевых понятий . . . . .	530
14.1.1. Разные подходы к ИИ . . . . .	530
14.1.2. Что делает глубокое обучение особенным среди других подходов к машинному обучению . . . . .	531
14.1.3. Как правильно воспринимать глубокое обучение . . . . .	531
14.1.4. Ключевые технологии . . . . .	533
14.1.5. Обобщенный процесс машинного обучения . . . . .	534
14.1.6. Основные архитектуры сетей . . . . .	535
14.1.7. Пространство возможностей . . . . .	540
14.2. Ограничения глубокого обучения . . . . .	542
14.2.1. Риск очеловечивания моделей глубокого обучения . . . . .	543
14.2.2. Автоматы и носители интеллекта . . . . .	546
14.2.3. Локальное и экстремальное обобщение . . . . .	548
14.2.4. Назначение интеллекта . . . . .	550
14.2.5. Восхождение по спектру обобщения . . . . .	551
14.3. Курс на увеличение универсальности в ИИ . . . . .	552
14.3.1. О важности постановки верной цели: правило выбора кратчайшего пути . . . . .	553
14.3.2. Новая цель . . . . .	555
14.4. Реализация интеллекта: недостающие ингредиенты . . . . .	557
14.4.1. Интеллект как чувствительность к абстрактным аналогиям . . . . .	557
14.4.2. Два полюса абстракции . . . . .	559
14.4.3. Недостающая половина картины . . . . .	563

14.5. Будущее глубокого обучения . . . . .	564
14.5.1. Модели как программы . . . . .	565
14.5.2. Сочетание глубокого обучения и синтеза программ . . . . .	566
14.5.3. Непрерывное обучение и повторное использование модульных подпрограмм . . . . .	569
14.5.4. Долгосрочная перспектива . . . . .	571
14.6. Как не отстать от прогресса в быстроразвивающейся области . . . . .	572
14.6.1. Практические решения реальных задач на сайте Kaggle . . . . .	572
14.6.2. Знакомство с последними разработками на сайте arXiv . . . . .	573
14.6.3. Исследование экосистемы Keras . . . . .	573
Заключительное слово . . . . .	574

*Моему сыну Сильвену: надеюсь,  
что когда-нибудь ты прочтешь эту книгу!*

# *Предисловие*

---

Если вы взяли в руки эту книгу, то, вероятно, наслышаны о недавнем небывалом успехе методики глубокого обучения в области искусственного интеллекта. Мы прошли путь от малопригодных реализаций компьютерного зрения и обработки естественного языка до высокопроизводительных систем, поставляемых в составе продуктов, которые вы используете каждый день. Последствия этого внезапного прогресса отразились почти на всех отраслях. Технологии глубокого обучения уже применяются для решения многих важных задач в медицине, сельском хозяйстве, автомобильной промышленности, образовании, прогнозировании стихийных бедствий и на производстве.

И все же, по моему мнению, глубокое обучение до сих пор находится в зачаточном состоянии. Реализована лишь малая часть его потенциала. Со временем оно найдет применение в каждой поставленной задаче, но это нескорый процесс, который наверняка займет несколько десятилетий.

Чтобы начать внедрение глубокого обучения во все необходимые задачи, мы должны сделать его доступным как можно большему числу людей, включая неспециалистов (которые не являются инженерами-исследователями или аспирантами). Раскрытие всего потенциала этой технологии требует полной ее демократизации. Сегодня мы находимся на пике исторического перехода, когда глубокое обучение выходит из академических лабораторий и отделов исследований крупных технологических компаний и становится неотъемлемой частью инструментария каждого разработчика — подобно тому как начинали распространяться веб-технологии в конце 1990-х. В 1998 году для создания сайта или приложения потребовалась бы небольшая команда инженеров. А теперь подобные продукты для своего бизнеса или сообщества может разработать любой желающий. В недалеком будущем специалисту хватит лишь базовых навыков программирования, чтобы воплотить свою идею интеллектуального приложения, обучающегося на данных.

Когда в марте 2015 года я выпустил первую версию Keras — фреймворка глубокого обучения, — я не задумывался о демократизации искусственного интеллекта (ИИ). К тому времени я уже несколько лет занимался исследованиями

в области машинного обучения и создал Keras как помощь в экспериментах. Однако начиная с 2015 года огромное число людей открыло для себя область глубокого обучения — и многие посчитали мой фреймворк неплохим подспорьем. Наблюдая за самыми неожиданными и довольно единственными способами использования Keras, я пришел к выводу, что мне нужно позаботиться о доступности ИИ. Я осознал: чем шире мы распространим эти технологии, тем ценнее они станут. Доступность была быстро определена как одна из главных целей Keras, и за несколько лет сообществу разработчиков удалось добиться фантастических результатов в этом направлении. Мы в буквальном смысле вручили технологию глубокого обучения сотням тысяч специалистов, а они, в свою очередь, воспользовались ею для решения важных задач, которые до недавнего времени считались нерешаемыми.

Данная книга — 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. В 2016–2017 годах я подготовил такой курс — и он нашел воплощение в первом издании этой книги, вышедшем в декабре 2017 года. Книга быстро стала бестселлером по машинному обучению — она разошлась тиражом более 50 000 экземпляров и была переведена на 12 языков, в том числе на русский (2018 год).

С момента выхода первого издания сфера глубокого обучения быстро развивалась: была выпущена версия TensorFlow 2, продолжила набирать популярность архитектура Transformer и т. д. Как результат, в конце 2019 года я решил обновить книгу. Сначала по простоте душевной я думал, что она изменится примерно наполовину и останется плюс-минус такого же объема, как и первое издание. Но после двух лет работы книга выросла на третью и обновилась почти на три четверти. Так что перед вами не просто исправленное и дополненное издание — это совершенно новая книга.

Я писал ее, стараясь максимально доступно объяснить идеи, лежащие в основе глубокого обучения, и их реализации. Это не значит, что я преднамеренно упрощал изложение — всецело уверен, что в теме глубокого обучения нет ничего сложного. Надеюсь, книга принесет вам пользу и поможет начать создавать интеллектуальные приложения для решения важных вам задач.

# *Благодарности*

---

Хочу поблагодарить сообщество Keras за помощь в создании этой книги. За последние шесть лет проект значительно вырос — в настоящее время он насчитывает несколько сотен разработчиков и более миллиона пользователей. Ваш вклад и отзывы помогли превратить Keras в то, чем он является сейчас.

Большое спасибо моей супруге за безграничную поддержку на протяжении всей работы над Keras и над этой книгой.

Благодарю компанию Google за поддержку Keras. Было очень приятно, когда там решили использовать мой проект 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 людей.

Хочу поблагодарить сотрудников издательства Manning, сделавших возможным выпуск этой книги: издателя Марджана Бейса и всех сотрудников редакторского и технического отделов, в том числе Майкла Стивенса, Дженнифер Стоут, Александра Драгосавлевича и многих других, чья работа осталась «за кадром».

Большое спасибо техническим рецензентам: Билли О'Каллагану, Кристиану Вайстаннеру, Конраду Тейлору, Даниэле Сапата Риеско, Дэвиду Джейкобсу, Эдмону Беголи, доктору Эдмунду Рональду, Хао Лю, Джареду Дункану, Ки Наму, Кену Фрикласу, Челлу Янссону, Милану Шаренацу, Нгуену Као, Никосу Канакарису, Оливеру Кортену, Раушану Джа, Саяку Полу, Серджио Гованни, Шашанку Поласу, Тодду Куку, Витону Витанису — и всем остальным, кто прислал свои замечания к рукописи этой книги.

Особое спасибо Фрэнсису Буонтемпо, выступившему в роли научного редактора, и Карстену Стробеку, выполнившему техническую редактуру книги.

---

<sup>1</sup> Открытая программная библиотека алгоритмов машинного обучения, разработанная компанией Google для решения задач построения и обучения нейронных сетей с целью автоматического поиска и классификации образов, качество которых достигает качества человеческого восприятия. — Примеч. пер.

# *O книге*

---

Книга написана для всех, кто хочет начать изучение технологии глубокого обучения с нуля или расширить уже имеющиеся знания. Инженеры, работающие в области машинного обучения, разработчики программного обеспечения и студенты найдут много ценного на страницах этого издания.

Технологии глубокого обучения будут описываться максимально доступно — мы начнем с самого простого, постепенно переходя к последним достижениям. Я старался найти баланс между теорией и практикой и избегать математических формул, предпочитая объяснять основные идеи с помощью фрагментов кода и интуитивно понятных моделей. Вы увидите множество 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 в качестве внутреннего механизма взяты фреймворк глубокого обучения Keras, написанный на Python, и библиотека TensorFlow 2. Они демонстрируют новейшие по состоянию на 2021 год приемы их использования.

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 в различных областях, от распознавания образов до обработки естественного языка: классификации образов, сегментирования изображений, временного прогнозирования, классификации текста, машинного перевода с одного языка на другой, генерации текста и многое другое.

## **КОМУ АДРЕСОВАНА ЭТА КНИГА**

Книга написана для людей с опытом программирования на Python, желающих познакомиться с машинным обучением в целом и глубоким обучением в частности. Однако она также может быть полезной для других читателей:

- если вы специалист по обработке и анализу данных, знакомый с машинным обучением, это издание 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, в книге вы найдете лучший интенсивный его курс;
- если вы аспирант, изучающий технологии глубокого обучения на занятиях, это издание станет практическим дополнением к учебным материалам, поможет лучше понять принцип действия глубоких нейросетей и познакомит с наиболее эффективными приемами.

Даже люди с техническим складом ума, которые не занимаются программированием регулярно, посчитают эту книгу полезной в качестве введения в базовые и продвинутые понятия глубокого обучения.

Для понимания кода примеров необходимо владеть языком Python на среднем уровне. Также не помешает знакомство с библиотекой NumPy, хотя это опционально. Опыт в машинном или глубоком обучении не является обязательным условием: книга раскрывает все необходимые основы. Не нужна какая-то особенная математическая подготовка — вполне достаточно знания математики на уровне средней школы.

## О ПРИМЕРАХ КОДА

Книга содержит множество примеров исходного кода как в листингах, так и внутри основного текста. В обоих случаях код набран **таким моноширинным шрифтом**, чтобы можно было выделить его на фоне других материалов.

Во многих случаях исходный код был переформатирован: добавлены разрывы строк и изменены размеры отступов, чтобы уместить код по ширине книжной страницы. Кроме того, комментарии из исходного кода удалялись, если он подробно описывается в тексте. Примечания в листингах дополняют описание в основном тексте, помогая выделить важные понятия.

Все примеры кода из этой книги доступны на сайте издательства Manning: <https://www.manning.com/books/deep-learning-with-python-second-edition>, а также в блокнотах Jupyter на GitHub: <https://github.com/fchollet/deep-learning-with-python-notebooks>. Их можно запускать с помощью Google Colaboratory — бесплатной среды для блокнотов Jupyter. Подключение к интернету и браузер — все, что вам нужно, чтобы начать знакомство с глубоким обучением.

## *Об авторе*

---



**Франсуа Шолле** является создателем Keras — одного из самых широко используемых фреймворков глубокого обучения. В настоящее время работает в Google, где разрабатывает программное обеспечение и возглавляет команду Keras. Кроме того, он занимается исследованиями в области абстрагирования, формализации рассуждений и обобщения в сфере искусственного интеллекта.

## *Иллюстрация на обложке*

---

Иллюстрация на обложке 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, опубликованной в Лондоне между 1757 и 1772 годами. На титульной странице указано, что это выполненная вручную каллиграфическая цветная гравюра, обработанная гуммиарабиком.

Томас Джейферис (1719–1771) носил звание географа 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 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 яркой индивидуальности и уникальности народов мира около 200 лет назад. С тех пор стиль одежды сильно изменился и исчезло разнообразие, характеризующее различные области и страны. Теперь трудно отличить по одежде даже жителей разных континентов. Если взглянуть на это оптимистично, мы пожертвовали культурной и внешней многогранностью в угоду более насыщенной личной жизни или в угоду многогранной и интересной интеллектуальной и технической деятельности.

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

## *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.

# 1

## Что такое глубокое обучение

### В этой главе

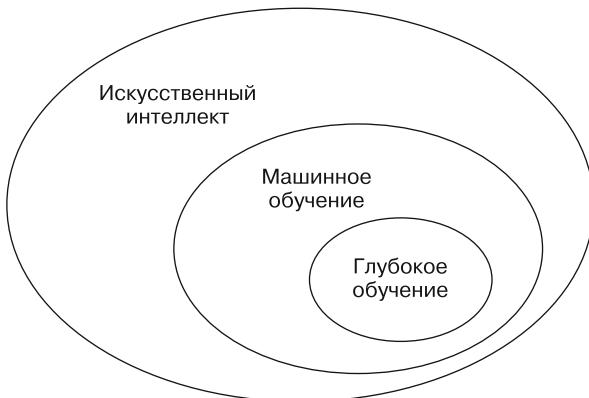
- ✓ Обобщенные определения основных понятий.
- ✓ История развития машинного обучения.
- ✓ Ключевые факторы роста популярности глубокого обучения и его потенциал в будущем.

За последние несколько лет тема искусственного интеллекта (ИИ) вызвала сильный резонанс 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 умных чат-ботов, беспилотных автомобилей и виртуальных помощников. Иногда будущее рисовали в мрачных тонах, а иногда изображали утопическим — где люди освобождены от рутинного труда, а основную работу выполняют роботы, наделенные искусственным интеллектом. Настоящему и будущему 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 в том числе от вас зависит, как оно в итоге будет выглядеть. Ведь, закончив чтение данной книги, вы вольетесь в ряды разработчиков систем ИИ. Поэтому давайте рассмотрим следующие вопросы. Чего уже достигло глубокое обучение? Насколько оно важно? В каком направлении пойдет его дальнейшее развитие? Можно ли верить поднятой шумихе?

Далее будет заложен 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 годах, когда группа энтузиастов из только зарождающейся области информатики задалась вопросом, можно ли заставить компьютеры «думать», что породило другие вопросы, ответы на которые мы ищем до сих пор.

Притом что многие из идей, легших в основу ИИ, появились за многие годы и даже десятилетия до этого, искусственный интеллект окончательно превратился в область исследований только в 1956 году, когда Джон Маккарти, тогда молодой доцент математического факультета в Дартмутском колледже, организовал летний семинар, откликнувшись на следующее предложение:

«Исследование должно базироваться на предположении, что каждый аспект обучения или любая другая особенность интеллекта в принципе может быть описана настолько точно, что на основе такого описания можно создать машину, моделирующую интеллект. Нужно попытаться найти способ заставить машины использовать язык, формировать абстракции и понятия, решать виды задач, которые ныне доступны только людям, и совершенствовать себя. Мы думаем, что в разрешении одной или нескольких из этих проблем можно добиться значительного прогресса, если специально подобранный комитет ученых будет совместно работать над ней в течение лета».

В конце лета семинар завершился, так и не дав полноценный ответ на загадку, которая была предметом исследований. Тем не менее он объединил многих будущих первопроходцев данной области и стал толчком интеллектуальной революции, продолжающейся по сей день.

Коротко ИИ можно определить так: *попытка автоматизации интеллектуальных задач, обычно выполняемых людьми*. Соответственно, ИИ — это область, охватывающая машинное и глубокое обучение, а также включающая многие подходы, с обучением не связанные. Имейте в виду, что до 1980-х годов в большинстве книг по искусственному интеллекту вообще отсутствовало такое понятие, как обучение. 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 искусственный интеллект уровня человека можно создать, если предоставить программисту достаточный набор явных правил для манипулирования знаниями. Этот поход, известный как *символический ИИ*, являлся доминирующей парадигмой ИИ с 1950-х до конца 1980-х годов. Пик его популярности пришелся на бум *экспертных систем* в 1980-х.

Символический ИИ прекрасноправлялся с четко определенными логическими задачами (такими как игра в шахматы). Но, как оказалось, это не работало для более сложных и менее четких случаев (например, для классификации изображений, распознавания речи и перевода на другие языки), ведь для их решения задать строгие правила невозможно. Поэтому на смену символическому ИИ пришел новый подход: *машинальное обучение*.

### **1.1.2. Машинное обучение**

В викторианской Англии жила леди Ада Лавлейс — друг и соратник Чарльза Бэббиджа, изобретателя *аналитической вычислительной машины* (первого известного механического компьютера). Несомненно, устройство опередило свое время, но в 1830-х и 1840-х годах оно не задумывалось как универсальный компьютер, потому что самой идеи универсальных вычислений еще не существовало. Машина просто давала возможность использовать механические операции для автоматизации некоторых вычислений из области математического анализа, что и обусловило такое ее название. Тем не менее она была более интеллектуальным потомком ранних механических вычислительных устройств, таких как суммирующая машина Паскаля («Паскалина») или счетчик шагов Лейбница — усовершенствованная версия «Паскалины». Разработанная Блезом Паскалем в 1642 году (в возрасте 19 лет!) «Паскалина» была первым в мире механическим арифмометром — она могла складывать, вычитать, умножать и даже делить.

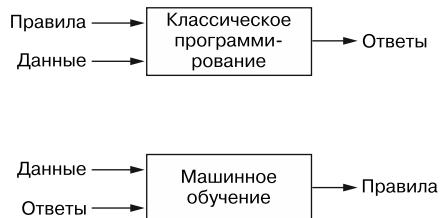
В 1843 году Ада Лавлейс заметила:

«Аналитическая машина не может создавать что-то новое. Она может делать все, что и мы... ее цель — лишь помогать нам осуществлять то, с чем мы уже хорошо знакомы».

Наблюдение леди Лавлейс остается поразительным даже спустя 179 лет. Сможет ли универсальный компьютер «создавать» что-нибудь свое, или он всегда будет просто выполнять операции, полностью понятные нам, людям? Сможет ли когда-нибудь породить какую-либо оригинальную мысль? Сможет ли учиться на собственном опыте? Сможет ли стать творцом?

Позднее пионер ИИ Аллан Тьюринг в своей знаменитой статье *Computing Machinery and Intelligence*<sup>1</sup> назвал это замечание «аргументом Ады Лавлейс»<sup>2</sup>. Там же он представил *тест Тьюринга*, а также перечислил основные идеи, которые могут привести к созданию ИИ<sup>3</sup>. Тьюринг придерживался весьма провокационного для того времени мнения, что компьютеры могут имитировать в принципе все аспекты человеческого интеллекта.

Обычно, чтобы заставить компьютер выполнять полезную работу, нужно создать *правила* — программу, которой нужно следовать, чтобы преобразовать входные данные в соответствующие ответы (точно так же, как леди Лавлейс записывала пошаговые инструкции для аналитической вычислительной машины). Машинное обучение меняет ситуацию: машина просматривает входные данные и соответствующие ответы и выясняет, какими должны быть правила (рис. 1.2). В машинном обучении система *обучается*, а не программируется явно. Ей передаются многочисленные примеры, имеющие отношение к данной задаче, а она находит там статистическую структуру, которая позволяет выработать соответствующие правила для решения этой задачи. Например, чтобы автоматизировать сортировку фотографий, сделанных в отпуске, можно передать системе машинного обучения множество примеров фото, уже отобранных людьми, — и система выучит статистические правила классификации конкретных материалов.



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

<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.

<sup>3</sup> Тест Тьюринга иногда интерпретировали буквально — как цель, которую должен достичь ИИ, — однако в действительности Тьюринг имел в виду лишь концептуальный прием в философской дискуссии о природе познания.

Расцвет машинного обучения начался только в 1990-х годах, но данное направление стало наиболее популярной и успешной частью ИИ — эта тенденция подкрепилась появлением быстродействующей аппаратуры и огромных наборов данных. Машинное обучение тесно связано с математической статистикой, но имеет несколько важных отличий — подобно тому как медицина связана с химией, но не может быть сведена только к ней, ведь включает свои особенные ответвления. В отличие от статистики машинное обучение обычно имеет дело с большими и сложными наборами данных (скажем, миллионы фотографий, каждая из которых состоит из десятков тысяч пикселей), к которым практически невозможно применить классические методы статистического анализа, например байесовские. Как результат, машинное, и в особенности глубокое, обучение не имеет мощной математической платформы и основывается почти исключительно на инженерных решениях. В отличие от теоретической физики или математики машинное обучение — это очень практическая сфера, основанная на эмпирических данных и сильно зависящая от достижений в области информатики и вычислительной техники.

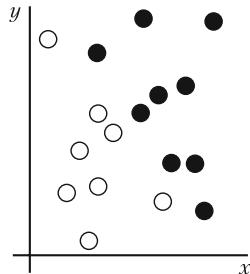
### **1.1.3. Изучение правил и представлений данных**

Чтобы дать определение *глубокому обучению* и понять разницу между этим и другими методами машинного обучения, сначала нужно узнать, что *делают* алгоритмы машинного обучения. Как отмечалось выше, машинное обучение выявляет правила решения задач обработки данных по примерам ожидаемых результатов. То есть нам нужны три составляющие:

- *контрольные входные данные* — например, если решается задача распознавания речи, такими данными могут быть файлы с записью речи разных людей. Если нужно классифицировать изображения, понадобятся соответствующие изображения;
- *примеры ожидаемых результатов* — в задаче распознавания речи это обычно транскрипции звуковых файлов, составленные людьми. При классификации изображений ожидаемым результатом могут быть теги: «собака», «кошка» и др.;
- *способ оценки качества работы алгоритма* — необходим для определения того, как сильно отклоняются результаты, возвращаемые алгоритмом, от ожидаемых. Оценка используется в качестве сигнала обратной связи для корректировки работы алгоритма. Этот этап корректировки мы и называем *обучением*.

Модель машинного обучения трансформирует контрольные входные данные в значимые результаты, «обучаясь» на известных примерах того и другого. То есть главной задачей машинного и глубокого обучения является *значимое преобразование данных*, или, иными словами, обучение *представлению* входных данных, приближающему нас к ожидаемому итогу.

Прежде чем двинуться дальше, давайте определим, что есть представление данных. По сути, это другой способ их *представления*, или *кодирования*. 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, «выбрать все красные пиксели» проще в RGB-изображениях, тогда как «сделать изображение менее насыщенным» быстрее в формате HSV. Главная задача моделей машинного обучения как раз заключается в поиске соответствующего представления входных данных – преобразований, которые сделают данные более пригодными для решения конкретной проблемы.



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

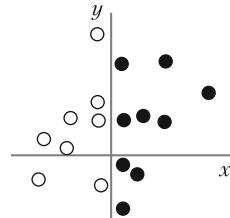
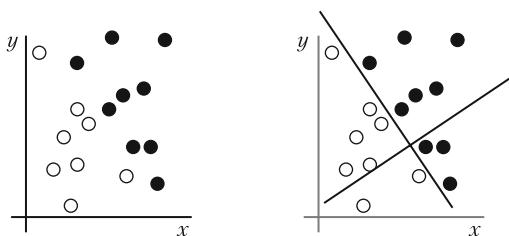
**Рис. 1.3.** Пример некоторых данных

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;
- мерой качества работы алгоритма может быть, например, процент правильно классифицированных точек.

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

1. Исходные данные    2. Изменение системы координат    3. Лучшее представление



**Рис. 1.4.** Изменение системы координат

Координаты наших точек в изменившейся системе координат можно назвать новым представлением данных. Причем более удачным! Задачу классификации данных «черный/белый» здесь можно свести к простому правилу: «черные точки имеют координату  $x > 0$ » или «белые точки имеют координату  $x < 0$ ». Это новое представление в сочетании с найденным правилом точно решает поставленную задачу.

В данном примере мы определили изменение координат вручную: использовали человеческий интеллект, чтобы придумать надлежащее представление данных. Этот подход можно с успехом применять в похожих простых задачах. Но смогли бы вы с такой же легкостью классифицировать изображения рукописных цифр? Получилось бы у вас явно сформулировать правила преобразования, которые подчеркнули бы разницу между шестеркой и восьмеркой, единицей и семеркой, написанными разными людьми?

Отчасти это возможно. Такие правила, как «количество замкнутых окружностей», или вертикальные и горизонтальные пиксельные гистограммы позволяют довольно точно различать рукописные цифры. Но отыскать подобные полезные представления вручную очень непросто. К тому же система, основанная на жестких правилах, очень хрупкая — просто кошмар для поддержки. Каждый раз, столкнувшись с нетипичным образцом почерка, нарушающим тщательно продуманные ранее протоколы, вам придется добавлять новые, не забывая при этом учитывать их взаимовлияние со всеми предыдущими.

Вы, наверное, думаете: если этот процесс такой болезненный, можно ли его автоматизировать? Что, если мы системно опробуем различные наборы представлений данных, сгенерированных автоматически, и правила, на них основанные, определяя наилучшие, базируясь на проценте правильно классифицированных цифр в некоторой первоначальной выборке? Это и будет самое настояще машинное обучение. *Обучение* в контексте машинного обучения описывает процесс автоматического поиска преобразований, создающих полезные представления определенных данных, который управляется сигналом обратной связи — представлениями, подчиненными более простым правилам решения поставленной задачи.

Алгоритмы машинного обучения обычно не выделяются чем-то особенным: они просто выполняют поиск в предопределенном наборе операций, который называют *пространством гипотез*. Например, в задаче классификации точек таким пространством будет пространство всех возможных преобразований двумерной системы координат.

То есть технически машинное обучение — это поиск значимого представления и правил по некоторым входным данным 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.

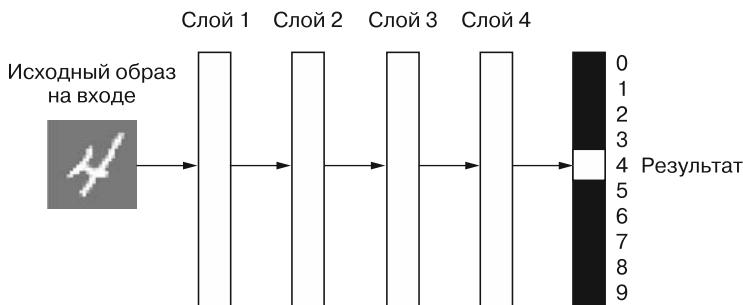
Теперь, когда вы знаете, что понимается под *обучением*, посмотрим, что особенного в *глубоком обучении*.

#### 1.1.4. «Глубина» глубокого обучения

Глубокое обучение — особый раздел машинного обучения, новый подход к поиску представления данных, делающий упор на изучении последовательных *слоев* (или *уровней*) все более значимых представлений. Под «глубиной» в глубоком обучении не подразумевается более детальное понимание, достигаемое этим подходом; идея заключается в создании многослойного представления. Поэтому подходящими названиями для этой области машинного обучения могли бы также служить *многослойное обучение* и *иерархическое обучение*. Число слоев, на которые делится модель данных, называют *глубиной* модели. Современное глубокое обучение часто вовлекает в процесс десятки и даже сотни последовательных слоев представления — все они автоматически определяются на основе обучающих данных. Тогда как другие подходы машинного обучения ориентированы на изучение всего одного-двух слоев; по этой причине их еще иногда называют *поверхностным обучением*.

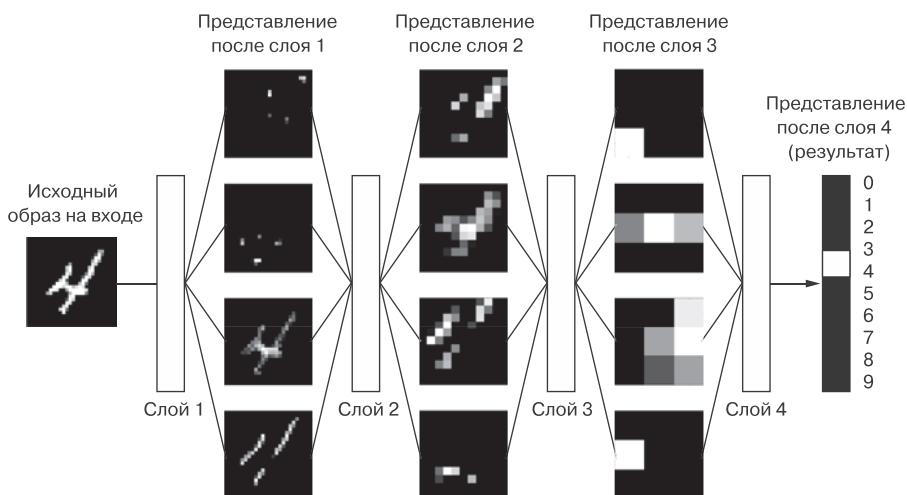
В глубоком обучении такие многослойные представления рассматриваются (почти всегда) с использованием *нейронных сетей* — моделей, структурированных в виде слоев, наложенных друг на друга. Термин «нейронная сеть» знаком нам из нейробиологии; хотя некоторые основополагающие идеи глубокого обучения действительно отчасти заимствованы из науки о мозге, его модели все же не являются моделями мозга. Нет никаких доказательств, что мозг реализует механизмы, подобные используемым в глубоком обучении. Вам могут встретиться научно-популярные статьи, где утверждается, что глубокое обучение работает подобно мозгу или моделирует работу мозга, но в действительности это не так. Было бы неправильно заставлять новичков в этой области думать, что глубокое обучение каким-то образом связано с нейробиологией. Забудьте всю эту туманную мистику про «как наш мозг»; забудьте также все, что читали о гипотетической связи между глубоким обучением и биологией. Намного продуктивнее считать данный метод математическим инструментом для изучения представлений данных.

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



**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 изображения и несущее все больше полезной информации. Глубокую сеть можно рассматривать как многоэтапную операцию очистки, во время которой информация проходит через последовательность фильтров и выходит из нее в очищенном (пригодном для решения поставленной задачи) виде.



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

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

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

Теперь вы понимаете, что суть машинного обучения заключается в преобразовании ввода (изображений) в результат путем исследования множества примеров входных данных и результатов. Вы также знаете, что глубокие нейронные сети выполняют при этом длинную последовательность простых преобразований (слоев) и обучаются этим преобразованиям на примерах. Давайте посмотрим, как именно проходит обучение.

То, что именно слой делает со своими входными данными, определяется его *весами*, которые фактически являются набором чисел. Выражаясь техническим языком, преобразование, реализуемое слоем, *параметризуется* весами (рис. 1.7). (Веса также иногда называют *параметрами слоя*.) В данном контексте *обучение* — это поиск набора значений весов всех слоев в сети, при котором сеть будет правильно отображать образцы входных данных в соответствующие им результаты. Но вот в чем дело: глубокая нейронная сеть может иметь десятки миллионов параметров. Поиск правильного значения для каждого из них может оказаться сложнейшей задачей, особенно если изменение значения одного параметра влияет на поведение всех остальных!

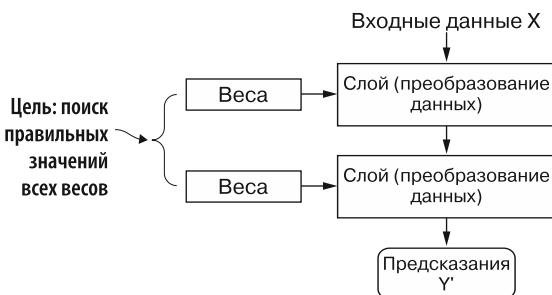
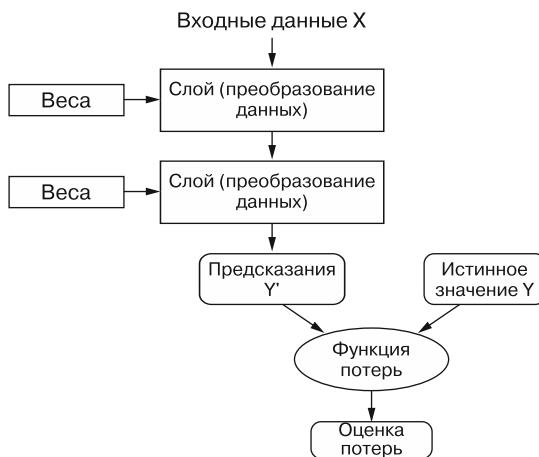


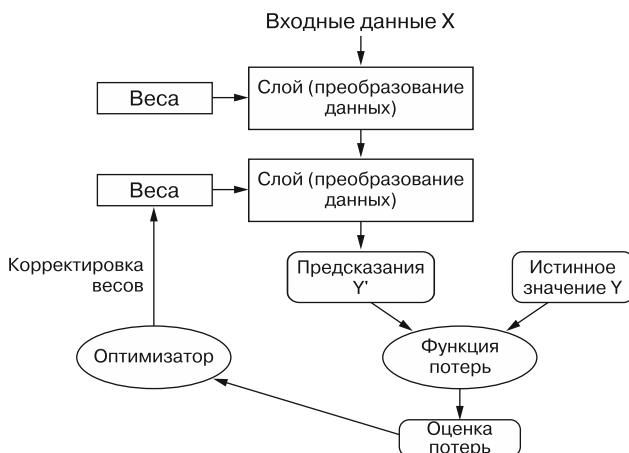
Рис. 1.7. Нейронная сеть параметризуется ее весами

Чтобы чем-то управлять, сначала нужно получить возможность наблюдать за этим. Чтобы управлять результатом работы нейронной сети, нужно измерить, насколько он далек от ожидаемого. Эту задачу решает *функция потерь* сети, называемая также *целевой функцией* или *функцией стоимости*. 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. 1.8.** Функция потерь оценивает качество результатов, производимых нейронной сетью

Основная хитрость глубокого обучения заключается в использовании этой оценки для корректировки значения 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.



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

Первоначально весам сети присваиваются случайные значения, то есть фактически сеть реализует последовательность случайных преобразований.

Естественно, получаемый ею результат далек от идеала, и оценка потерь при этом очень 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 в 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 решении задач восприятия и обработки естественного языка — задач, кажущихся натуральными и понятными для человека, но долгое время не дававшихся компьютерам.

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

- 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 Assistant 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. С его помощью мы достигли значительных успехов в решении широкого круга задач, непосильных для компьютеров еще несколько лет назад, — в автоматической расшифровке десятков тысяч древних рукописей, хранящихся в Апостольском архиве Ватикана; определении и классификации

болезней растений в полевых условиях с использованием обычного смартфона; интерпретации данных медицинских снимков для онкологов или радиологов; прогнозировании таких стихийных бедствий, как наводнения, ураганы или даже землетрясения, и т. д. С каждым новым достижением мы приближаемся к эпохе, когда глубокое обучение будет нам полезно во всех сферах человеческой деятельности — в науке, медицине, производстве, энергетике, транспорте, разработке программного обеспечения, сельском хозяйстве и даже в художественном творчестве.

### **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 притом, что многие значительные варианты применения (такие как автопилоты для автомобилей) находятся практически на заключительной стадии реализации, другие (полноценные диалоговые системы, перевод между произвольными языками и понимание естественного языка на уровне человека), скорее всего, еще долго 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 на какое-то время замедлится.

Такое уже происходило раньше. ИИ пережил две волны оптимистического подъема, за которыми следовал спад, сопровождаемый разочарованиями, скептицизмом и, как результат, снижением финансирования. Все началось с символического ИИ в 1960-х. В те годы давались весьма многообещающие прогнозы его развития. Один из самых известных пионеров и сторонников символического ИИ Марвин Мински в 1967 году заявил: «В течение поколения... проблема создания “искусственного интеллекта” будет практически решена». Три года спустя, в 1970 году, он сделал более точное предсказание: «Через 3–8 лет у нас появится машина с интеллектом среднего человека». В 2021-м это достижение все еще кажется далеким — пока нам сложно предугадать, сколько времени уйдет на это, — но в 1960-х и в начале 1970-х некоторые эксперты (как и многие люди ныне) полагали, что будущее находится прямо за углом. Несколько лет спустя из-за не оправдавшихся высоких ожиданий исследователи и правительственные фонды отвернулись от этой области — так началась первая зима ИИ (метафора вполне уместна: все это происходило вскоре после начала холодной войны).

Данный спад был не последним. В 1980-х интерес к символическому ИИ снова возрос благодаря буму экспертных систем в крупных компаниях. Первые успехи вызвали волну инвестиций — и отделы ИИ, занимающиеся разработкой экспертных систем, начали появляться в корпорациях по всему миру. 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 к ней снова начал падать. Так началась вторая зима ИИ.

В настоящее время мы подходим к третьему циклу разочарования в ИИ, но пока 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 несмотря на наши, возможно, нереалистичные ожидания на ближайшую перспективу, долгосрочная картина выглядит весьма ярко. Мы только начинаем применять глубокое обучение в решении многих важных проблем, от постановки медицинских диагнозов до усовершенствования цифровых помощников. В последние пять лет исследования в области ИИ продвигались удивительно быстро во многом благодаря высокому уровню финансирования, никогда прежде не наблюдавшемуся в недолгой истории ИИ, но пока слишком малому, чтобы этот прогресс получил свое воплощение в продуктах и процессах, формирующих наш мир. Большинство результатов исследований в глубоком обучении пока не нашли практической реализации, по крайней мере в полном спектре задач, где эта технология могла быть полезна. Ваш доктор и ваш бухгалтер пока не используют ИИ. Вы сами в повседневной жизни, скорее всего, с ним не сталкиваетесь. Конечно, вы задаете простые вопросы своему смартфону и получаете разумные ответы, вам попадаются весьма полезные рекомендации при выборе товаров на Amazon, а по фразе «день рождения» вы можете быстро найти в Google Photos фотографии с праздника вашей дочери, который был в прошлом месяце. Это, несомненно, большой шаг вперед. Но такие инструменты лишь дополняют нашу жизнь. ИИ еще не занял в ней центральное место.

Сейчас трудно поверить, что ИИ может оказать значительное влияние на наш мир, потому что он еще не развернулся во всю ширь. Так же и в 1995 году трудно было поверить в будущее влияние интернета — большинство людей не понимало, какое отношение к ним может иметь Всемирная сеть и как она 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 проследит за здоровьем. Он доставит продукты к вашей двери и отвезет вас из пункта А в пункт Б. Это будет ваш интерфейс мира, все более усложняющегося и наполняющегося 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 нашего общества и нашей жизни, как и интернет сегодня.

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

## **1.2. ЧТО БЫЛО ДО ГЛУБОКОГО ОБУЧЕНИЯ: КРАТКАЯ ИСТОРИЯ МАШИННОГО ОБУЧЕНИЯ**

Глубокое обучение привлекло общественное внимание и инвестиции, невиданные 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 правильный инструмент — иногда просто недостаточно данных для глубокого обучения, а 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, начнете воспринимать все задачи машинного обучения как гвозди. Единственный способ не попасть в эту ловушку — познакомиться с другими подходами и практиковать их по необходимости.

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* — 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 данной категории является наивный байесовский алгоритм.

Наивный байесовский алгоритм — это вид классификатора машинного обучения, основанный на применении теоремы Байеса со строгими (или «наивными» — отсюда и название алгоритма) предположениями о независимости входных данных.

Настоящая 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 — это все, что нужно было для использования наивных байесовских классификаторов.

С байесовским 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. 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 сегодня полностью вытеснены актуальными вариантами (о которых рассказывается на страницах этой книги), но вам будет полезно знать и о корнях глубокого обучения. Основные идеи нейронных сетей в упрощенном виде были исследованы еще в 1950-х годах. 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 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 в Bell Labs объединил ранние идеи сверточных нейронных сетей и обратного распространения ошибки и использовал их для решения задачи распознавания рукописных цифр. Получившаяся в результате нейронная сеть была названа *LeNet* и была внедрена почтовой службой США в 1990-х для автоматического распознавания почтовых индексов на конвертах.

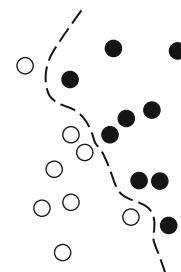
## 1.2.3. Ядерные методы

Хотя первый успех нейронных сетей в 1990-х и привлек к ним внимание исследователей, новый разработанный подход к машинному обучению — ядерные методы (*kernel methods*) — быстро отправил нейронные сети обратно в не-бытие. *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 и Коринной Кортес в начале 1990-х в Bell Labs и опубликована в 1995 году<sup>1</sup>, хотя прежняя линейная формулировка была обнародована Вапником и Алексеем Червоненкисом еще в 1963 году<sup>2</sup>.

Метод опорных векторов — это алгоритм классификации, предназначенный для поиска хороших «решающих границ», разделяющих два класса (рис. 1.10). Он выполняется 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*. Он позволяет обобщить классификацию новых образцов, не принадлежащих обучающему набору данных.



**Рис. 1.10.** Решающая граница

Методика отображения данных в пространство более высокой размерности, где задача классификации становится проще, может хорошо выглядеть на бумаге, но на практике часто оказывается трудноразрешимой. 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 определять координаты точек в этом пространстве не требуется; достаточно вычислить расстояния между парами точек — эффективно это можно сделать с помощью функции ядра. *Функция ядра* — это незатратная вычислительная операция, отображающая любые две точки из исходного пространства и вычисляющая расстояние между ними в целевом пространстве представления, полностью минуя явное вычисление нового представления. Функции ядра обычно определяются вручную, а не извлекаются из 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

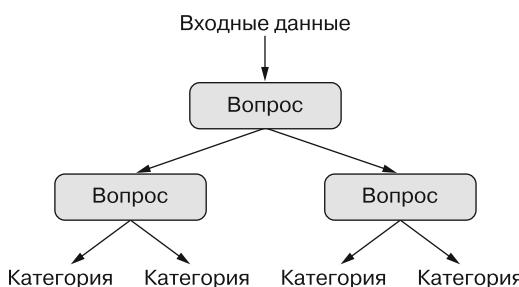
<sup>1</sup> Vapnik V., Cortes C. Support-Vector Networks // Machine Learning 20, no. 3 (1995): 273–297.

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

и поддающихся серьезному математическому анализу, что сделало его понятным и легко интерпретируемым. Благодаря этому SVM приобрел чрезвычайную популярность на долгие годы. Однако он оказался трудноприменимым к большинством наборов данных и не давал хороших результатов для таких задач, как классификация изображений. Поскольку метод опорных векторов является поверхностным методом, для его использования в распознавании данных требуется сначала вручную определить представительную выборку (этот шаг называется *конструированием признаков*), что сопряжено со сложностями и чревато ошибками. Например, у вас не получится применить SVM для классификации рукописных цифр, имея только исходные изображения; вам придется сначала вручную найти такие представления изображений, которые упростят решение задачи, например вышеупомянутые гистограммы распределения пикселей.

#### 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 году им часто отдавали предпочтение перед ядерными методами.



**Рис. 1.11.** Дерево решений: обучаемыми параметрами являются вопросы о данных. Таким вопросом мог бы быть, например: «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. Он создает большое количество специализированных деревьев решений и в последующем объединяет выдаваемые ими результаты. Случайные леса применимы к широкому кругу проблем — можно сказать, они почти всегда являются оптимальным алгоритмом для любых задач поверхностного машинного обучения. Когда

в 2010 году был запущен известный конкурсный веб-сайт Kaggle (<http://kaggle.com>), посвященный машинному обучению, случайные леса быстро набрали там популярность и удерживали свои позиции, 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, специализирующихся на устранении слабых мест в предыдущих моделях. Применительно к деревьям решений настоящий прием позволяет получить модели, которые в большинстве случаев превосходят случайные леса, сохраняя аналогичные свойства. На сегодняшний день это один из лучших алгоритмов (если не *самый* лучший) для решения задач, не связанных с распознаванием. Наряду с глубоким обучением данный прием на сайте Kaggle находится среди наиболее используемых.

### 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 в этой сфере, стали добиваться важных успехов: группы Джекфри Хинтона из Университета Торонто, Йошуа Бенгио из Университета Монреяля, Яна Лекуна из Нью-Йоркского университета и исследователи в научно-исследовательском институте искусственного интеллекта IDSIA в Швейцарии.

В 2011 году Ден Киресан из IDSIA выиграл академический конкурс по классификации изображений с использованием 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 по крупномасштабному распознаванию образов (ImageNet Large Scale Visual Recognition Challenge, или кратко ILSVRC). ImageNet предложило очень сложное на то время задание, заключающееся в делении цветных изображений с высоким разрешением на 1000 разных категорий после обучения по выборке, включающей 1,4 миллиона изображений. В 2011 году модель-победитель, основанная на классических подходах к распознаванию образов, показала точность лишь 74,3%<sup>1</sup>. В 2012 году команда Алекса Крижевски, в которой советником был Джекфри Хинтон, достигла точности 83,6% — значительный прорыв. С тех пор каждый год первые позиции в этом соревновании занимают глубокие сверточные нейронные сети. В 2015 году точность модели-победителя составляла 96,4% — и задача классификации на ImageNet была сочтена решенной полностью.

---

<sup>1</sup> Точность оценивается как частота выбора моделью из основных своих предполагаемых ответов правильного (на каждый из 1000 вопросов в случае ImageNet).

Начиная с 2012 года глубокие сверточные нейронные сети (convnets) перешли в разряд передовых алгоритмов для всех проблем распознавания образов; в целом, они с успехом могут использоваться в любых задачах распознавания. На крупных конференциях по распознаванию образов, проводившихся после 2015 года, 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 — например, в обработке естественного языка. В широком круге вопросов оно полностью заменило метод опорных векторов и деревья решений. 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 было принято решение перейти на использование глубоких нейронных сетей на основе Keras из-за лучшей производительности и простоты их обучения на больших наборах данных.

### **1.2.6. Отличительные черты глубокого обучения**

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

Более ранние методы машинного обучения — методы поверхностного обучения — включали 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 приходилось прилагать большие усилия, чтобы привести исходные данные к виду, более пригодному для обработки этими методами, в том числе вручную улучшать слой представления своих данных. Это называется *конструированием признаков*. Глубокое обучение, напротив, полностью автоматизирует этот шаг: применяя методы глубокого обучения, все признаки извлекаются за один проход, без необходимости конструировать их вручную. Процесс машинного обучения, таким образом, значительно упростился: часто сложный и многоступенчатый конвейер оказалось возможным заменить единственной простой сквозной моделью глубокого обучения.

Вы можете поинтересоваться: если суть рассматриваемого предмета заключается в получении нескольких последовательных слоев представлений, можно ли

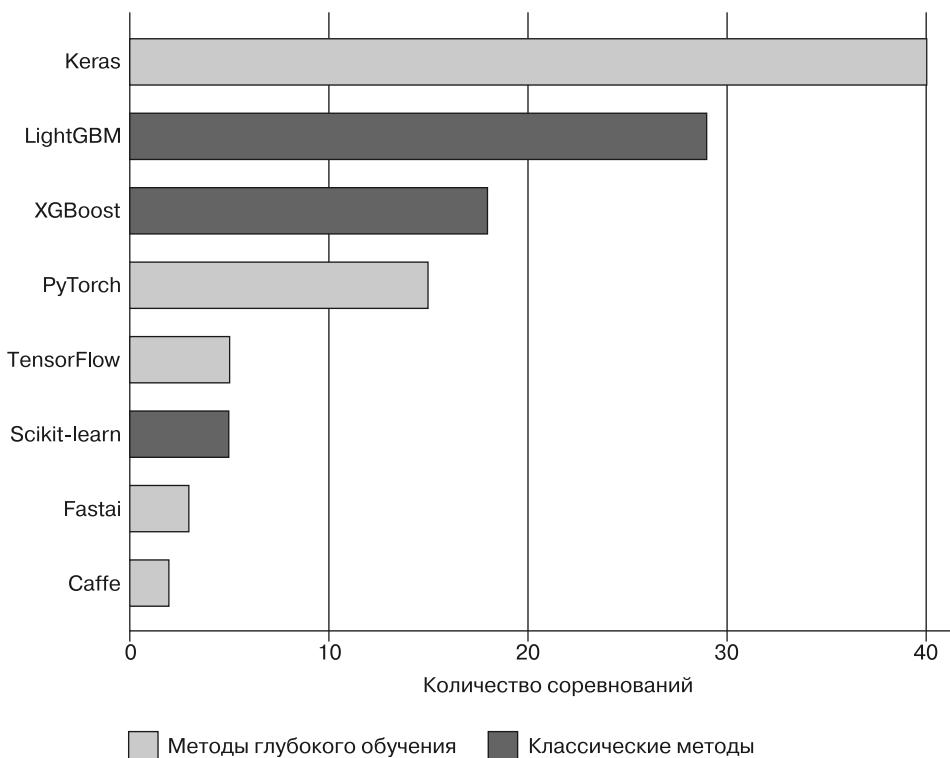
многократно применить методы поверхностного обучения для имитации эффекта глубокого обучения? На практике последовательное использование методов поверхностного обучения дает быстрое уменьшение отдачи, поскольку оптимальный слой первого представления в трехслойной модели не является оптимальным первым слоем в однослойной или двухслойной модели. Особенность преобразования в глубоком обучении состоит в том, что модель может исследовать все слои представления *вместе* и одновременно, а не последовательно (последовательное исследование также называют *жадным*). При совместном изучении, когда модель изменяет один из своих внутренних признаков, все прочие признаки, зависящие от него, в соответствии с этим корректируются автоматически, без вмешательства человека. Все контролируется единственным сигналом обратной связи: каждое изменение в модели служит конечной цели. Это намного эффективнее, чем жадно накладывать поверхностные модели друг на друга, потому что позволяет исследовать 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**

Отличный способ получить представление о текущей ситуации в использовании алгоритмов и инструментов машинного обучения — это конкурсный сайт Kaggle. Благодаря соревновательному характеру (в некоторых конкурсах участвуют тысячи соискателей, а призы составляют миллионы долларов США) и широкому разнообразию задач машинного обучения Kaggle помогает реально оценить, какие существуют подходы и насколько они успешны. Так какой же алгоритм уверенно выигрывает состязания? Какими инструментами пользуются победители?

В начале 2019 года у команд, которые начиная с 2017 года попадали в пятерку лучших в любом из соревнований Kaggle, поинтересовались, какой основной программный инструмент они использовали (рис. 1.12). Как оказалось, ведущие команды отдавали предпочтение методам глубокого обучения (обычно с применением библиотеки Keras) или деревьям с градиентным бустингом (как правило, с использованием библиотеки LightGBM или XGBoost).



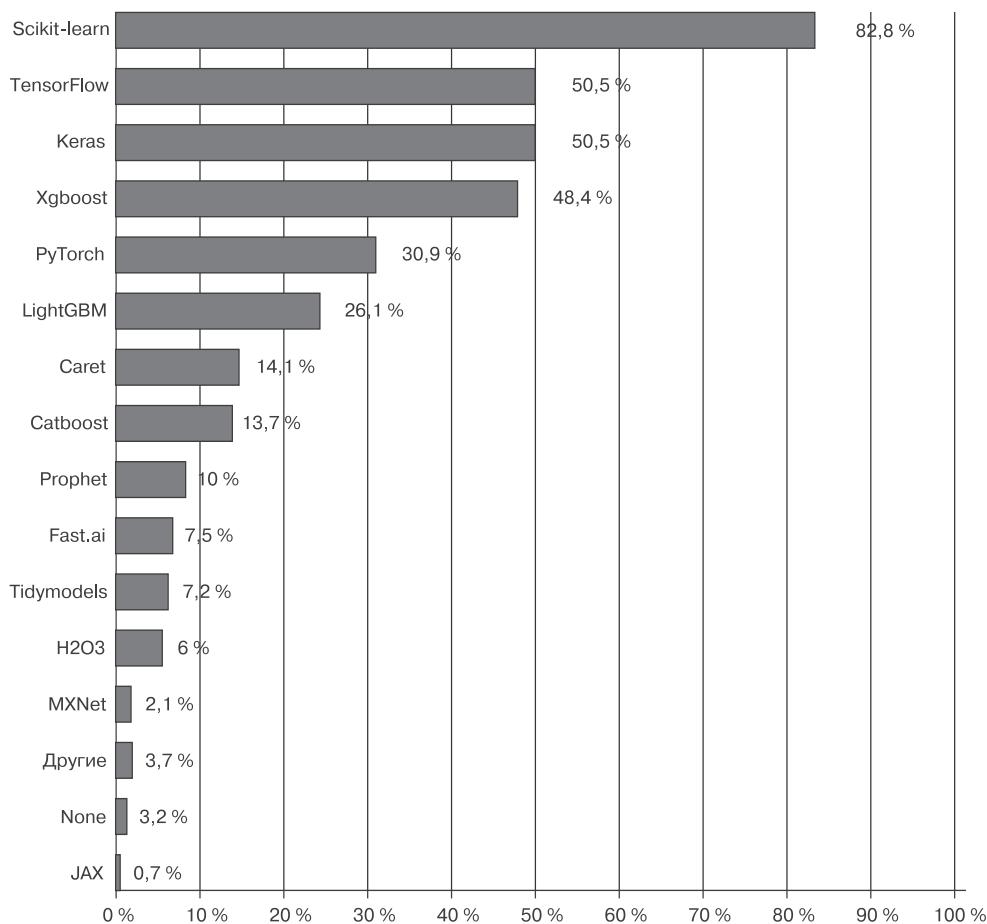
**Рис. 1.12.** Инструменты машинного обучения, использовавшиеся командами, которые участвовали в конкурсах Kaggle

Впрочем, авторов исследования интересуют не только победители. Kaggle ежегодно проводит опрос среди специалистов по всему миру, профессионально занимающихся машинным обучением и обработкой данных. В нем участвуют десятки тысяч респондентов, поэтому он считается одним из самых надежных источников информации о состоянии отрасли. На рис. 1.13 показан процент использования различных программных инструментов машинного обучения.

С 2016 по 2020 год в индустрии машинного обучения и обработки данных гла-венствовали два подхода: метод градиентного бустинга и глубокое обучение. Метод градиентного бустинга, в частности, использовался для решения задач, где присутствовали структурированные данные, тогда как глубокое обучение применялось для решения задач распознавания, таких как классификация изображений.

Приверженцы градиентного бустинга почти всегда используют Scikit-learn, XGBoost или LightGBM. А подавляющее большинство специалистов, практикующих глубокое обучение, предпочитают библиотеку Keras, обычно в комбинации

с фреймворком TensorFlow. Эти инструменты имеют одну общую черту — все они являются библиотеками на языке Python, широко используемым для решения задач машинного обучения и анализа данных.



**Рис. 1.13.** Использование инструментов в индустрии машинного обучения и обработки данных (источник: [www.kaggle.com/kaggle-survey-2020](http://www.kaggle.com/kaggle-survey-2020))

Чтобы добиться успеха в применении машинного обучения, следует уделить особое внимание двум методам: методу градиентного бустинга (для задач поверхностного обучения) и глубокому обучению (для задач распознавания). В техническом плане это означает, что вы должны владеть тремя библиотеками — Scikit-learn, XGBoost и Keras, — занимающими доминирующее положение в конкурсах на сайте Kaggle. Как только вы взяли в руки данную 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 модифицировался. 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.

На протяжении 1990-х и 2000-х годов данные и оборудование были действительно узким местом. Но в это же время случилось следующее: интернет значительно развелся, 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. Сейчас на ноутбуке можно запускать небольшие модели глубокого обучения, тогда как 25 лет назад это в принципе было невозможно.

Однако типичные модели глубокого обучения, используемые для распознавания образов или речи, требуют вычислительной мощности на порядок 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. Теперь несколько GPU могут заменить мощные кластеры на обычных процессорах в различных задачах с возможностью массового распараллеливания вычислений (в том числе начиная с физического моделирования). Глубокие нейронные сети, выполняющие в основном умножение множества маленьких матриц, также допускают высокую степень распараллеливания, поэтому ближе к 2011 году некоторые исследователи начали писать CUDA-реализации нейронных сетей. Одними из первых стали Дэн Кайесан<sup>1</sup> и Алекс Крижевски<sup>2</sup>.

Таким образом игровая индустрия субсидировала создание суперкомпьютеров для следующего поколения приложений искусственного интеллекта. Действительно, иногда крупные достижения начинаются с игр. Современный графический процессор NVIDIA Titan RTX, в конце 2019 года стоивший 2500 долларов США, способен выдать пиковую производительность 16 терафлопс с одинарной точностью (16 триллионов операций в секунду с числами типа `float32`). Это почти в 500 раз больше производительности самого быстрого по состоянию на 1990 год суперкомпьютера Intel Touchstone Delta. Графическому процессору Titan RTX требуется всего несколько часов для обучения модели ImageNet, выигравшей конкурс ILSVRC в 2012–2013 годах. Между тем большие компании совершенствуют модели глубокого обучения на кластерах, состоящих из сотен GPU.

Более того, индустрия глубокого обучения вышла за рамки GPU и инвестировала средства в развитие еще более специализированных, эффективных процессоров для глубокого обучения. В 2016 году на ежегодной конференции Google I/O компания Google продемонстрировала свой проект тензорного процессора (Tensor Processing Unit, TPU) с новой архитектурой, предназначенного для использования в глубоких нейронных сетях, намного более производительного и энергоэффективного, чем топовые модели GPU.

В 2020 году было представлено третье поколение карты TPU с вычислительной мощностью 420 терафлопс. Это в 10 000 раз больше мощности Intel Touchstone Delta 1990 года.

Данные карты TPU предназначены для сборки крупномасштабных конфигураций, называемых блоками или подами (pods). Один блок (1024 карты TPU) имеет максимальную производительность 100 петафлопс. Для сравнения — это

---

<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>.

около 10 % пиковой вычислительной мощности современного крупнейшего суперкомпьютера IBM Summit в Национальной лаборатории Ок-Риджа, который состоит из 27 000 графических процессоров NVIDIA и имеет пиковую вычислительную мощность около 1,1 экасофлопса.

### 1.3.2. Данные

Иногда ИИ называют новой индустриальной революцией. И если глубокое обучение — ее паровой двигатель, то данные — это уголь: сырье, питающее наши интеллектуальные машины, без которого невозможно движение вперед. В добавок к экспоненциальному росту емкости устройств хранения информации, наблюдавшемуся в последние 20 лет (согласно закону Мура), перемены в игровом мире вызвали бурное развитие 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, классифицированные пользователями, стали золотой жилой для разработчиков моделей распознавания образов. То же можно сказать о видеороликах на YouTube. А «Википедия» теперь считается 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 1,4 миллиона изображений, классифицированных вручную на 1000 категорий (каждое изображение отнесено только к одной категории). Но особенной коллекции делает не только ее огромный размер, но и ее применение во время ежегодных соревнований<sup>1</sup>.

Как показывает пример Kaggle, публичные конкурсы — отличный способ мотивации исследователей и инженеров преодолевать все новые и новые рубежи. Наличие общих критериев оценки достижений участников значительно помогло недавнему росту глубокого обучения, подчеркнув его преимущества перед классическими подходами к машинному обучению.

### 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 довольно неглубокими, имеющими один или два слоя представления; в связи с этим они не могли противостоять более совершенным

<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).

поверхностным методам, таким как метод опорных векторов и случайные леса. Ключевой проблемой было *распространение градиента* через глубокие пакеты слоев. Сигнал обратной связи, используемый для обучения нейронных сетей, по мере увеличения количества слоев затухал.

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 по-слойного обучения (от которого быстро отказались);
- улучшенные *схемы оптимизации*, такие как RMSProp и Adam.

Только когда эти усовершенствования позволили создавать модели с десятью слоями и более, глубокое обучение получило свое развитие. А в 2014, 2015 и 2016 годах были открыты еще более продвинутые способы распространения градиента, такие как пакетная нормализация, обходные связи и отделимые свертки.

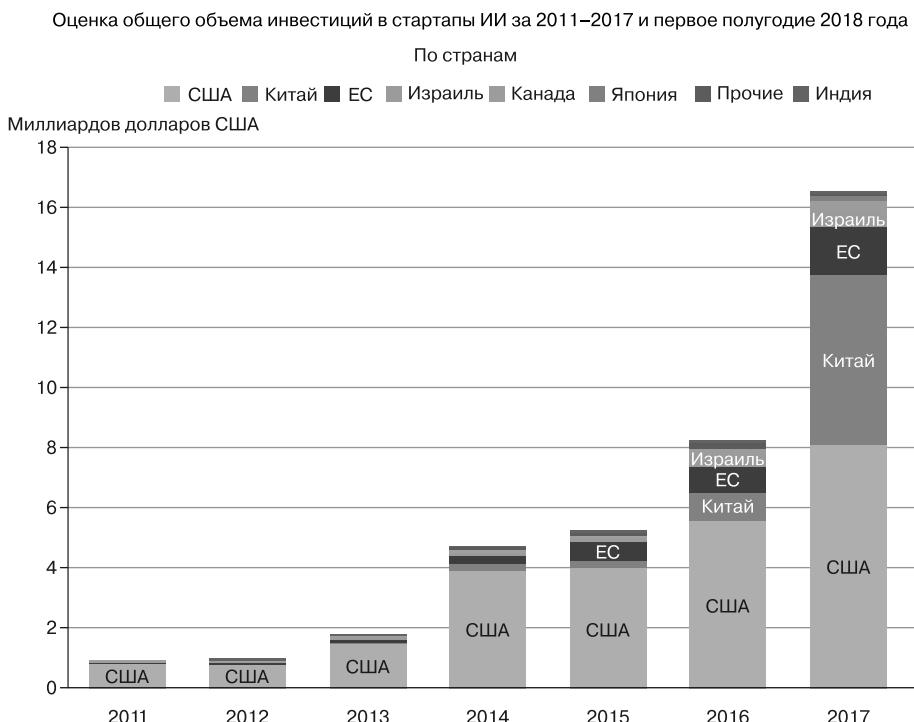
В настоящее время мы можем обучать с нуля модели с произвольной глубиной. Это открыло возможность использования чрезвычайно больших моделей, обладающих значительной презентативной силой, то есть способных кодировать обширнейшие пространства гипотез. Чрезвычайная масштабируемость — одна из определяющих характеристик современного глубокого обучения. Архитектуры крупномасштабных моделей, включающие десятки слоев и десятки миллионов параметров, позволили достичь важных рубежей в области распознавания образов (архитектуры ResNet, Inception или Xception) и в обработке естественного языка (большие архитектуры на основе Transformer: BERT, GPT-3 или XLNet).

### 1.3.4. Новая волна инвестиций

Как отметили ведущие исследователи, в 2012–2013 годах глубокое обучение вывело на новый современный уровень 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 по всему миру составили меньше одного миллиарда долларов — эти деньги почти полностью ушли на практическое применение методов поверхностного машинного обучения. К 2015 году вложения превысили пять миллиардов, а в 2017 достигли ошеломляющих 16 миллиардов (рис. 1.14). За эти несколько лет появились сотни

стартапов, пытающихся извлечь выгоду из поднявшейся шумихи. Между тем крупные компании, такие как Google, Amazon и Microsoft, инвестировали деньги в исследования, проводившиеся внутренними подразделениями, и объемы этих инвестиций почти наверняка превысили вложения венчурного капитала.



**Рис. 1.14.** Оценка ОЭСР общего объема инвестиций в стартапы ИИ  
(источник: <http://mng.bz/zGN6>)

Машинное обучение — и глубокое обучение в частности — заняло центральное место в стратегии продуктов этих технологических гигантов. В конце 2015 года генеральный директор Google Сундар Пичаи 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>.

Благодаря волне инвестиций менее чем за десять лет число людей, работающих над глубоким обучением, увеличилось с нескольких сотен до десятков тысяч, а прогресс в исследованиях достиг небывалого уровня.

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

### 1.3.5. Демократизация глубокого обучения

Одним из ключевых факторов, обусловивших приток новых лиц в глубокое обучение, стала демократизация инструментов, используемых в данной области. На начальном этапе глубокое обучение требовало значительных знаний и опыта программирования на C++ и владения CUDA, чем могли похвастаться очень немногие. В настоящее время для исследований в области глубокого обучения достаточно базовых навыков программирования на Python. Это вызвано прежде всего развитием Theano и позднее TensorFlow (двух фреймворков для Python, реализующих операции с тензорами, которые поддерживают автоматическое дифференцирование и значительно упрощают реализацию новых моделей), а также появлением дружественных библиотек (например, Keras), которые делают работу с глубоким обучением таким же простым делом, как манипулирование кубиками лего. После выхода в 2015 году библиотека Keras быстро была принята за основу многими командами новых стартапов, 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-инвесторов, и для исследователей? Или это просто увлечение, которое не продлится долго? Будем ли мы использовать глубокие нейронные сети через 20 лет?

Глубокое обучение имеет несколько свойств, которые подтверждают и укрепляют его статус «революции в ИИ». Возможно, нейронные сети исчезнут через два десятилетия, но все, что останется после них, будет прямым наследником современного глубокого обучения и его основных идей. Эти важнейшие идеи 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 5–6 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 осуществлять этот процесс на наборах данных произвольного размера (единственным узким местом является объем доступной 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 совершенствоваться на дополнительных данных без полного перезапуска, что делает их пригодными для непрерывного и продолжительного обучения — это чрезвычайно важно для очень больших промышленных моделей. Кроме того, подобные модели можно перенацеливать и, соответственно, задействовать 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 применить 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 2016 году, когда шла работа над первым изданием этой книги, я утверждал, что глубокое обучение все еще находится в первой половине сигмоиды и в следующие несколько лет ожидается гораздо больший прогресс в преобразованиях. Это подтвердилось на практике: в 2017 и 2018 годах наблюдался рост популярности моделей глубокого обучения на основе Transformer, используемых для обработки естественного языка и совершивших революцию в данной области. В то же время глубокое обучение продолжало обеспечивать устойчивый прогресс в области распознавания образов и речи. В 2021 году оно, похоже, перешло во вторую половину сигмоиды. Все еще можно ожидать значительного развития в предстоящие годы, но, скорее всего, мы уже вышли из начальной фазы стремительного прогресса.

Я чрезвычайно рад, что технологии глубокого обучения находят применение во все более широком круге задач — на самом деле их потенциальный список бесконечен. Глубокое обучение — это все еще революция, и потребуется много лет, чтобы полностью реализовать его потенциал.



# Математические основы нейронных сетей

## В этой главе

- ✓ 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 данной 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. Наиболее точным и однозначным описанием математической операции является выполняющий ее код.

Чтобы вам проще было разобраться с тензорами и градиентным спуском, мы начнем 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 освоите математическую теорию, на которой основывается глубокое обучение, и будете готовы погрузиться в изучение Keras и TensorFlow в главе 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 здесь поймете. Но пусть вас это не пугает. В следующей главе мы рассмотрим и подробно объясним каждый элемент в примере. Поэтому не волнуйтесь, если какие-то шаги покажутся странными или похожими на магию. В конце концов, мы должны с чего-то начать.

Перед нами стоит задача: реализовать классификацию черно-белых изображений рукописных цифр ( $28 \times 28$  пикселей) по десяти 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 годах. «Решение» задачи MNIST можно рассматривать как своеобразный аналог Hello World в глубоком обучении — часто это первое действие, которое выполняется для уверенности, 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.



Рис. 2.1. Образцы изображений MNIST

### ПРИМЕЧАНИЕ

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

Не пытайтесь сразу же воспроизвести пример на своем компьютере. Чтобы его опробовать, нужно сначала установить библиотеку Keras — а это будет рассмотрено в главе 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 tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Здесь `train_images` и `train_labels` — это *обучающий набор*, то есть данные, на которых модель обучается. 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`. Сеть обучится подбирать правильные метки для изображений. А затем мы предложим ей 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 tensorflow import keras  
from tensorflow.keras import layers  
model = keras.Sequential(  
    [layers.Dense(512, activation="relu"),  
     layers.Dense(10, activation="softmax")  
)
```

Основным строительным блоком нейронных сетей является *слой*. Слой можно рассматривать как фильтр для данных: он принимает их и выводит в некоторой более полезной форме. В частности, слои извлекают *представления* из входных 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 — слоев — все более тонкой работы с данными.

В нашем случае сеть состоит из последовательности двух слоев `Dense`, которые являются тесно связанными (их еще называют *полносвязными*) нейронными слоями. Второй (и последний) слой — это десятипеременный слой *классификации softmax*, возвращающий массив с десятью оценками вероятностей (в сумме дающих 1). Каждая оценка определяет вероятность принадлежности текущего изображения к одному из десяти классов цифр.

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

- *оптимизатор* — механизм, с помощью которого сеть будет обновлять себя, опираясь на наблюдаемые данные и функцию потерь;
- *функцию потерь* — определяет, как сеть должна оценивать качество своей работы на обучающих данных и, соответственно, корректировать ее в правильном направлении;
- *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

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Перед обучением мы выполним предварительную обработку данных, преобразовав 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
```

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

**Листинг 2.5.** Обучение («адаптация») модели

```
>>> model.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. Мы быстро достигли точности 0,989 (98,9 %).

Теперь у нас есть обученная модель, которую можно использовать для прогнозирования вероятностей принадлежности *новых* цифр к классам — изображений, которые не входили в обучающую выборку, как те из контрольного набора.

**Листинг 2.6.** Использование модели для получения предсказаний

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
```

Каждое число в элементе массива с индексом *i* соответствует вероятности принадлежности изображения цифры `test_digits[0]` к классу *i*.

Наивысшая оценка вероятности (0,99999106 — почти 1) для этого тестового изображения цифры находится в элементе с индексом 7, то есть согласно нашей модели — перед нами изображение цифры 7:

```
>>> predictions[0].argmax()
7
>>> predictions[0][7]
0.99999106
```

Прогноз можно проверить по массиву меток:

```
>>> test_labels[0]
7
```

В целом, насколько хорошо справляется наша модель с классификацией прежде не встречавшихся ей цифр? Давайте проверим, вычислив среднюю точность по всему контрольному набору изображений.

**Листинг 2.7.** Оценка качества модели на новых данных

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

Точность на контрольном наборе составила 97,8 % — немного меньше, чем на обучающем (98,9 %). Эта разница демонстрирует пример *переобучения* (overfitting), когда модели машинного обучения показывают точность на новом наборе данных худшую, чем на обучающем. Переобучение будет основной темой главы 3.

На этом мы завершаем наш первый пример — вы только что увидели, как создать и обучить нейронную сеть классификации рукописных цифр, написав меньше 15 строк кода на Python. Далее я подробнее расскажу обо всех встретившихся здесь деталях и поясню происходящее за кулисами. Вы узнаете о тензорах, объектах хранения данных в сети; об операциях с тензорами, выполняемых слоями; о градиентном спуске, позволяющем сети совершенствоваться 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, 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, обычно числовых. Проще говоря, 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 *осью*).

### 2.2.1. Скаляры (тензоры нулевого ранга)

Тензор, содержащий единственное число, называется *скаляром* (скалярным тензором, 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, 7])
>>> x
array([12, 3, 6, 14, 7])
>>> x.ndim
1
```

Этот вектор содержит пять элементов и потому 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 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 тензор второго ранга, 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 более высоких 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.

## 2.2.5. Ключевые атрибуты

Тензор определяется тремя ключевыми атрибутами, такими как:

- *количество осей (ранг)* — например, трехмерный тензор имеет три оси, а матрица — две. В библиотеках для Python, таких как NumPy или TensorFlow, данный атрибут имеет имя `ndim`;
- *форма* — кортеж целых чисел, описывающих количество измерений на каждой оси тензора. Например, матрица в предыдущем примере имеет форму  $(3, 5)$ , а тензор третьего ранга —  $(3, 3, 5)$ . Вектор имеет форму с единственным элементом, например  $(5,)$ , тогда как у скаляра форма пустая —  $()$ ;
- *тип данных (обычно в библиотеках для Python ему дается имя dtype)* — это тип данных, содержащихся в тензоре; например, тензор может иметь тип `float16`, `float32`, `float64`, `uint8` и др. В TensorFlow можно также встретить тензоры типа `string`.

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

```
from tensorflow.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`:

```
>>> train_images.ndim
3
```

его форму:

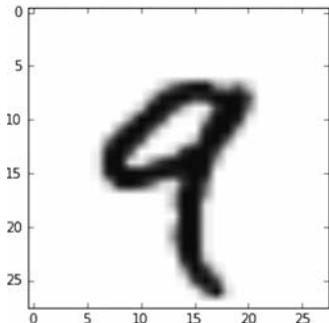
```
>>> train_images.shape
(60000, 28, 28)
```

и тип данных, заглянув в атрибут `dtype`:

```
>>> train_images.dtype  
uint8
```

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

Попробуем отобразить четвертую цифру из этого тензора, использовав библиотеку Matplotlib (известная библиотека на Python для визуализации данных, входящая в состав пакета Colab) (рис. 2.2).



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

#### Листинг 2.8. Вывод четвертой цифры на экран

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

Естественно, что этому изображению соответствует метка — целое число 9:

```
>>> train_labels[4]  
9
```

### 2.2.6. Манипулирование тензорами с помощью NumPy

В предыдущем примере мы *выбрали* конкретную цифру на первой оси, используя синтаксис `train_images[i]`. Операция выбора конкретного элемента в тензоре называется *получением среза тензора*. 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 с формой (90, 28, 28):

```
>>> my_slice = train_images[10:100]  
>>> 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, :, :] ← Эквивалентно предыдущему примеру
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28] ← Также эквивалентно предыдущему примеру
>>> 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:]
```

Также допускается использовать отрицательные индексы. Как и отрицательные индексы в списках на Python, они будут откладываться от конца текущей оси. Например, обрезать все изображения, оставив только квадрат  $14 \times 14$  пикселей в центре, можно следующим образом:

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

## 2.2.7. Пакеты данных

В общем случае первая 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. 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 (с индексом 0) называют *осью пакетов* или *измерением пакетов*. Данная 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, перечислю несколько примеров тензоров с данными, которые могут встретиться вам в будущем. Данные, которыми вам придется манипулировать, почти всегда будут относиться к одной из таких категорий, как:

- *векторные данные* — двумерные тензоры с формой (*образцы, признаки*), где каждый образец — это вектор числовых атрибутов (*«признаков»*);
- *временные ряды или последовательности* — трехмерные тензоры с формой (*образцы, метки\_времени, признаки*), где каждый образец является последовательностью (*длиной метки\_времени*) векторов признаков;
- *изображения* — четырехмерные тензоры с формой (*образцы, высота, ширина, цвет*), где каждый образец является двумерной матрицей пикселей, а каждый пиксель представлен вектором со значениями *«цвета»*;
- *видео* — пятимерные тензоры с формой (*образцы, кадры, высота, ширина, цвет*), где каждый образец является последовательностью (*длина равна значению кадры*) изображений.

### 2.2.9. Векторные данные

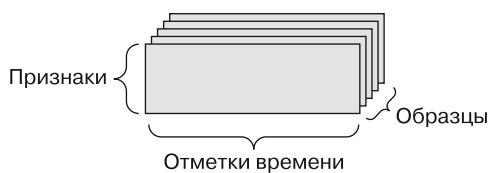
Наиболее часто встречающаяся форма данных. В таких наборах каждый образец может быть представлен вектором, а пакет соответственно — двумерным тензором (то есть массивом векторов), где первая ось — это *ось образцов*, а вторая — *ось признаков*.

Рассмотрим два примера.

- Актуарный набор данных с информацией о людях, где для каждого человека указываются возраст, пол 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 значениями (по одному счетчику на каждое слово из словаря), соответственно весь набор данных, описывающий 500 документов, можно сохранить в 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 сохранить 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

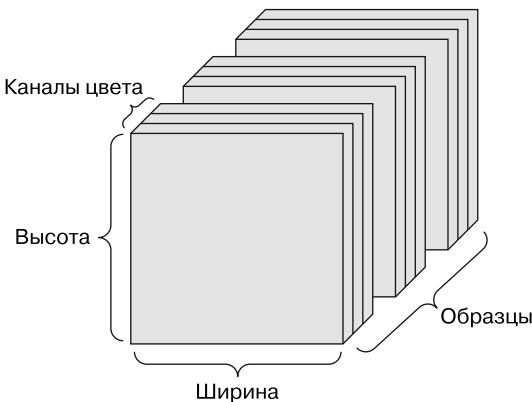
В соответствии с соглашениями ось времени всегда является второй осью (осью с индексом 1). Рассмотрим несколько примеров.

- 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 — матрицей с формой  $(390, 3)$  (где 390 — длительность торгового дня в минутах), а данные за 250 дней — трехмерным тензором формы  $(250, 390, 3)$ . В таком 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 притом, 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» (которое все больше теряет популярность).

По соглашению «канал следует последним» ось цвета помещается в конец: (образцы, высота, ширина, цвет). По соглашению «канал следует первым» ось цвета помещается сразу после оси пакетов: (образцы, цвет, высота, ширина). При следовании соглашению «канал следует первым» предыдущие примеры выглядели бы так:  $(128, 1, 256, 256)$  и  $(128, 3, 256, 256)$ . Библиотека Keras поддерживает оба формата.

## 2.2.12. Видео

Видеоданные — один из немногих типов данных, для хранения которых требуется 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 поместить 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 четыре 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, а для всего тензора соответственно 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* (или *тензорных функций*), применяемых к тензорам с числовыми данными. Например, тензоры можно складывать, перемножать и т. д.

В нашем первом примере мы создали модель, 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 матрицу и возвращает другую матрицу — новое представление исходного тензора. В данном случае функция имеет следующий вид (где  $W$  — это матрица, а  $b$  — вектор; оба значения являются атрибутами слоя):

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

Давайте развернем ее. Здесь у нас имеются три операции с тензорами:

- скалярное произведение (`dot`) исходного тензора `input` и тензора с именем `W`;
- сложение (`+`) получившейся матрицы и вектора `b`;
- операция `relu`:  $\text{relu}(x)$  — эквивалентна операции `max(x, 0)`; название `relu` происходит от английского *rectified linear unit* (блок линейной корректировки).

### ПРИМЕЧАНИЕ

Даже притом, что в этом разделе очень часто используются выражения из линейной алгебры, вы не найдете здесь математических формул. 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 применять код, использующий NumPy и TensorFlow.

### 2.3.1. Поэлементные операции

Операции `relu` и сложение — это *поэлементные операции*: то есть такие, которые применяются к каждому отдельному элементу в тензоре. Они поддаются масштабному распараллеливанию (*векторизация* — термин пришел из архитектуры *векторного процессора* суперкомпьютера периода 1970–1990). 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 перезапись
    for i in range(x.shape[0]): исходного тензора
        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 перезапись
    for i in range(x.shape[0]): исходного тензора
        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 C.

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

```
import numpy as np ← Поэлементное сложение
z = x + y
z = np.maximum(z, 0.) ← Поэлементная операция relu
```

Давайте измерим, насколько этот процесс «мгновенный»:

```
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.)
print("Took: {:.2f} s".format(time.time() - t0))
```

У меня эта операция выполнилась за 0,02 секунды, тогда как предыдущей наивной реализации потребовалось 2,45 секунды:

```
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {:.2f} s".format(time.time() - t0))
```

Аналогично при запуске на графическом процессоре код TensorFlow выполняет поэлементные операции с применением полностью векторизованных реализаций CUDA, которые максимально эффективно используют архитектуру графического процессора с высокой степенью параллелизма.

### 2.3.2. Расширение

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

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 (так называемые *оси расширения*), чтобы значение его атрибута `ndim` соответствовало значению этого же атрибута большего тензора.
2. Меньший тензор копируется в новые оси до полного совпадения с формой большего тензора.

Рассмотрим конкретный пример. Допустим, у нас имеются тензоры  $X$  с формой  $(32, 10)$  и  $y$  с формой  $(10,)$ :

```
import numpy as np
X = np.random.random((32, 10))           ← X — матрица случайных чисел с формой (32, 10)
y = np.random.random((10,))               ← y — вектор случайных чисел с формой (10,)
```

Сначала нужно добавить первую пустую ось в вектор  $y$ , чтобы привести его форму к виду  $(1, 10)$ :

```
y = np.expand_dims(y, axis=0) ← Теперь у имеет форму (1, 10)
```

Затем скопируем  $y$  по этой новой оси 32 раза, чтобы в результате получился тензор  $Y$  с формой  $(32, 10)$ , где  $Y[i, :] = y$  для  $i$  в диапазоне  $\text{range}(0, 32)$ .

```
Y = np.concatenate([y] * 32, axis=0) ← Скопировать у 32 раза вдоль оси 0,
                                         чтобы получить Y с формой (32, 10)
```

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
    assert len(y.shape) == 1.           ← тензор NumPy
    assert x.shape[1] == y.shape[0]     ← Убедиться, что y —
                                         вектор NumPy

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

Прием расширения в общем случае можно использовать 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 показывает применение поэлементной операции `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,
                                         имеющий форму (64, 3, 32, 10)
y = np.random.random((32, 10))           ← y — 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, оператором `*`), — наиболее общая и наиболее полезная операция с тензорами.

Поэлементное произведение в NumPy выполняется с помощью функции `np.dot` (потому что в математике тензорное произведение обозначают точкой ( $\bullet$ )):

```
x = np.random.random((32,))
y = np.random.random((32,))
z = np.dot(x, y)
```

В математике скалярное произведение обозначается точкой ( $\bullet$ ):

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

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

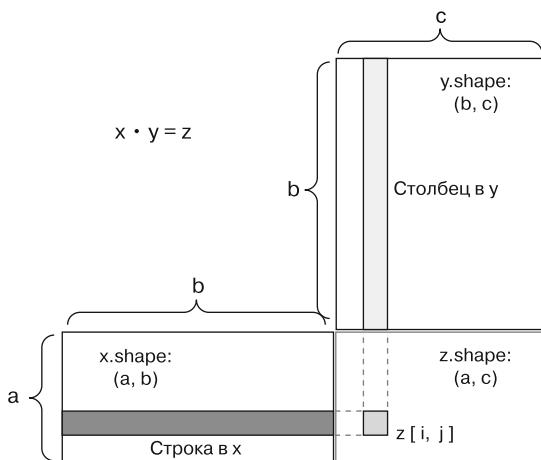
```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]): ← Убедиться, что x и y —
        for j in range(y.shape[1]): ← матрицы NumPy
            row_x = x[i, :] ← Первое измерение x должно
            column_y = y[:, j] ← совпадать с нулевым измерением y!
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

← Обход строк в x... ← ...и столбцов в y

Эта операция вернет  
матрицу заданной формы  
с нулевыми элементами

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

`x`, `y` и `z` изображены на рис. 2.5 в виде прямоугольников (буквально — таблиц элеменитов). Число строк в `x` и столбцов в `y` должно совпадать, соответственно, ширина `x` должна равняться высоте `y`. Если вы будете создавать новые алгоритмы машинного обучения, вам часто придется рисовать подобные диаграммы.



**Рис. 2.5.** Диаграмма скалярного произведения матриц

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:

$$\begin{aligned} (a, b, c, d) \cdot (d,) &\rightarrow (a, b, c) \\ (a, b, c, d) \cdot (d, e) &\rightarrow (a, b, c, e) \end{aligned}$$

и т. д.

### 2.3.4. Изменение формы тензора

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

```
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 новой формы имеет такое же количество элементов, что и исходный. Чтобы было понятнее, рассмотрим несколько простых примеров:

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

```
>>> 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) | Создаст матрицу с формой (300, 20),  
>>> print(x.shape) | заполненную нулями  
(20, 300)
```

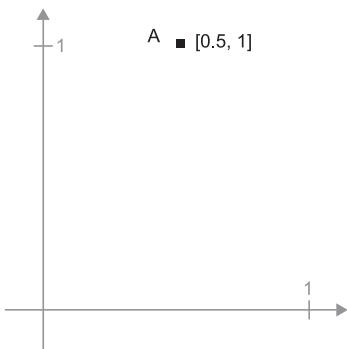
### 2.3.5. Геометрическая интерпретация операций с тензорами

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:

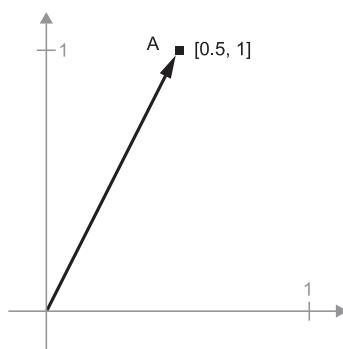
```
A = [0.5, 1]
```

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.

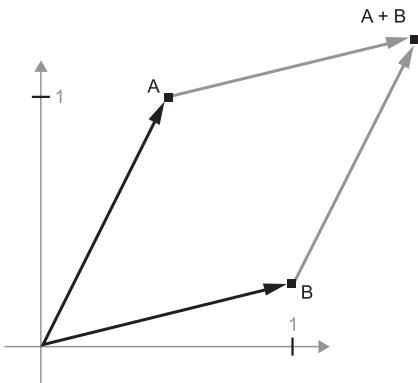
Добавим новый вектор  $B = [1, 0.25]$  и сложим его с предыдущим. Чтобы получить результирующий вектор, представляющий сумму двух исходных векторов, достаточно перенести начало одного вектора в конец другого (рис. 2.8). Как видите, операция прибавления вектора  $B$  к вектору  $A$  заключается в переносе точки  $A$  в новое место, при этом расстояние и направление переноса от исходной точки  $A$  определяются вектором  $B$ . Если применить ту же операцию к группе точек на плоскости («объекту»), можно создать копию целого объекта в новом месте (рис. 2.9). То есть тензорное сложение представляет операцию *параллельного переноса объекта* (без его искажения) на определенное расстояние в определенном направлении.



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

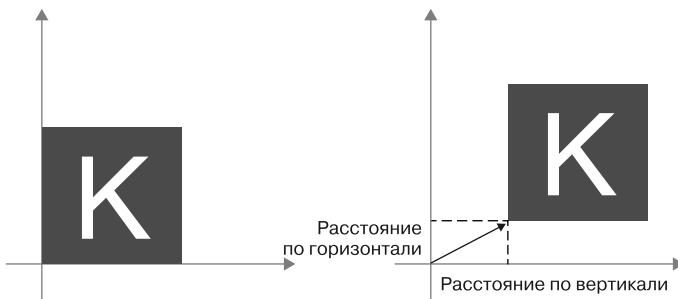


**Рис. 2.7.** Вектор в двумерном пространстве, изображенный в виде стрелки



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

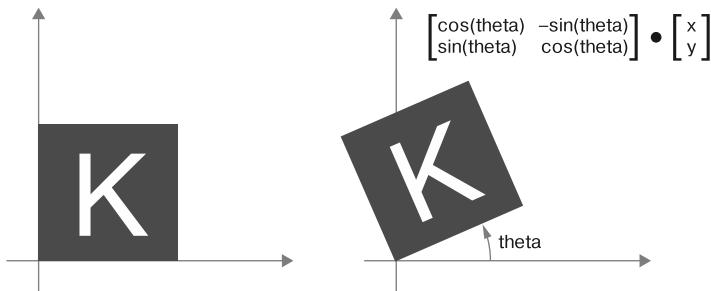
$$\begin{bmatrix} \text{Расстояние по горизонтали} \\ \text{Расстояние по вертикали} \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$



**Рис. 2.9.** Сложение векторов как параллельный перенос в двумерном пространстве

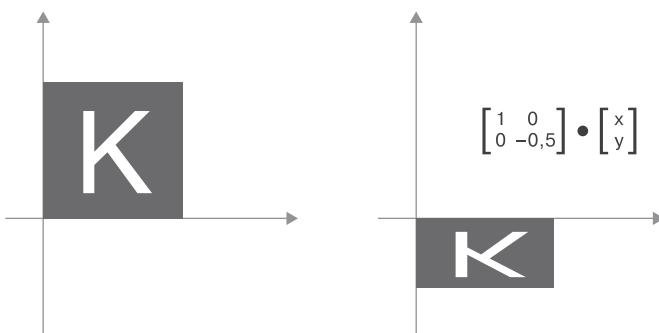
Элементарные геометрические операции, такие как параллельный перенос, поворот, масштабирование, наклон и т. д., можно выразить в виде операций с тензорами. Вот несколько примеров:

- *параллельный перенос*: как показано выше, добавление вектора к точке переместит ее на фиксированное расстояние в фиксированном направлении; при применении к набору точек (например, к двумерному объекту) эта операция называется параллельным переносом (рис. 2.9);
- *поворот*: поворот двумерного вектора на угол  $\theta$  против часовой стрелки (рис. 2.10) выражается как скалярное произведение с матрицей  $R = [u, v]$  размером  $2 \times 2$ ,  $R = [[\cos(\theta), -\sin(\theta)], [\sin(\theta), \cos(\theta)]]$ ;

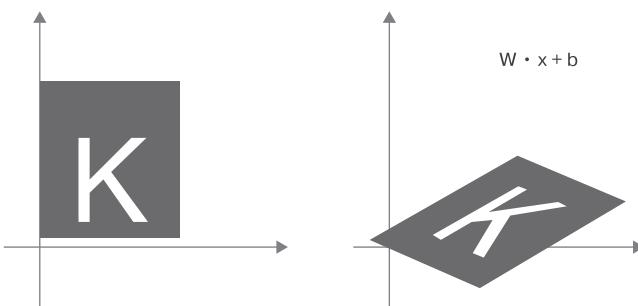


**Рис. 2.10.** Поворот двумерного вектора (против часовой стрелки) как скалярное произведение

- *масштабирование*: масштабирование изображения по вертикали и горизонтали (рис. 2.11) можно осуществить с помощью скалярного произведения с матрицей  $2 \times 2 S = [[\text{масштаб\_по\_горизонтали}, 0], [0, \text{масштаб\_по\_вертикали}]]$  (обратите внимание, что такие матрицы называются диагональными, поскольку имеют ненулевые коэффициенты только на главной диагонали, идущей от верхнего левого угла к нижнему правому);
- *линейное преобразование*: скалярное произведение с произвольной матрицей реализует линейное преобразование. Обратите внимание, что *масштабирование* и *поворот*, перечисленные выше, по определению являются линейными преобразованиями;
- *аффинное преобразование*: аффинное преобразование (рис. 2.12) — это комбинация линейного преобразования (путем скалярного произведения с некоторой матрицей) и параллельного переноса (путем сложения векторов). Как вы, наверное, уже заметили, именно это преобразование,  $y = W \bullet x + b$ , реализует слой Dense! Полносвязанный слой Dense без функции активации является аффинным слоем;

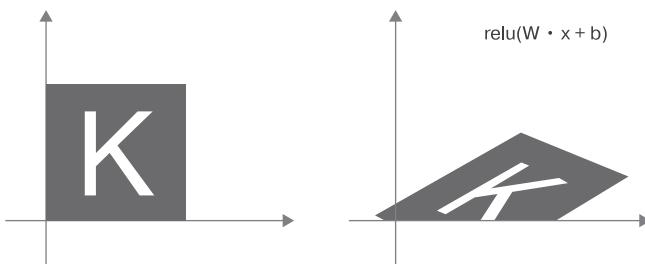


**Рис. 2.11.** Двумерное масштабирование как скалярное произведение



**Рис. 2.12.** Аффинное преобразование на плоскости

- *полносвязанный слой (Dense) с активацией relu:* аффинные преобразования обладают одним важным свойством — при многократном их применении в результате все равно получается аффинное преобразование (то есть можно сразу взять одно это суммирующее преобразование). Рассмотрим пример с двумя аффинными преобразованиями:  $\text{affine2}(\text{affine1}(x)) = W2 \bullet (W1 \bullet x + b1) + b2 = (W2 \bullet W1) \bullet x + (W2 \bullet b1 + b2)$ . Это аффинное преобразование, в котором линейная часть представлена матрицей  $W2 \bullet W1$ , а часть, отвечающая за параллельный перенос, — это вектор  $W2 \bullet b1 + b2$ . Как следствие, многослойная нейронная сеть, полностью состоящая из слоев Dense без активаций, будет эквивалентна одному слою Dense. Значит, данная «глубокая» нейронная сеть является всего лишь замаскированной линейной моделью! Вот почему нужны функции активации, такие как `relu` (ее действие показано на рис. 2.13). Благодаря им можно создать цепочку слоев Dense для реализации очень сложных нелинейных геометрических преобразований и получить богатое пространство гипотез для глубоких нейронных сетей. Этую идею мы подробно рассмотрим в следующей главе.

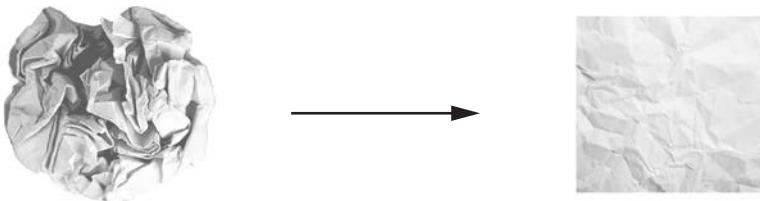


**Рис. 2.13.** Аффинное преобразование с последующим применением функции активации relu

### 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 в трехмерном пространстве полезно представить следующий мысленный образ. Вообразите два листа цветной бумаги: один красного цвета и другой — синего. Положите их друг на друга. Теперь сомните 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. 2.14.** Разглаживание смятого комка исходных данных

Разглаживание комка бумаги — вот в чем суть машинного обучения: в поиске ясных представлений для сложных, перемешанных данных. Сейчас у вас должно сложиться достаточно полное понимание, почему глубокое обучение преуспевает в этом: оно использует последовательное разложение сложных геометрических преобразований в длинную цепь простых — почти так же, как поступает человек, разворачивая смятый лист. Каждый слой в глубоком обучении применяет преобразование, которое немного распутывает данные, а использование множества слоев позволяет работать с очень сложными данными.

## 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(input, W) + b)
```

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

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

Ниже перечислены шаги, выполняемые в так называемом *цикле обучения*, который повторяется необходимое количество раз.

1. Извлекается пакет обучающих экземпляров  $x$  и соответствующих целей  $y_{\text{true}}$ .
2. Модель обрабатывает пакет  $x$  (этот шаг называется *прямым проходом*) и получает пакет предсказаний  $y_{\text{pred}}$ .
3. Вычисляются потери модели на пакете, дающие оценку несовпадения между  $y_{\text{pred}}$  и  $y_{\text{true}}$ .
4. Веса модели корректируются так, чтобы немного уменьшить потери на этом пакете.

В конечном итоге получается модель, имеющая очень низкие потери на обучении наборе данных: несовпадение предсказаний  $y_{pred}$  с ожидаемыми целями  $y_{true}$  малое. Модель «научилась» отображать входные данные в правильные конечные значения. Со стороны все это может походить на волшебство, однако, если разобрать процесс на мелкие шаги, он выглядит очень просто.

Шаг 1 несложный — это просто операция ввода/вывода. Шаги 2 и 3 — всего лишь применение нескольких операций с тензорами, и вы сможете реализовать их, опираясь на полученные в предыдущем разделе знания. Наиболее запутанным выглядит шаг 4: корректировка весов сети. Как по отдельным весам в сети узнать, должен ли некоторый коэффициент увеличиваться или уменьшаться и насколько?

Одно из простейших решений — заморозить все веса, кроме одного, и попробовать применить разные его значения. 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.

Однако подобный подход крайне неэффективен, поскольку требует выполнять два прямых прохода (что довольно затратно) для каждого отдельного веса (которых очень много, обычно тысячи, а иногда и до нескольких миллионов). К счастью, есть более оптимальное решение: *градиентный спуск*.

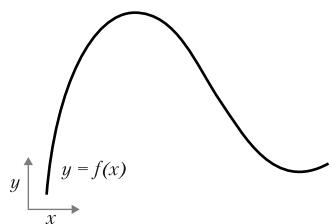
Градиентный спуск — метод оптимизации, широко применимый в современных нейронных сетях. Суть его заключается в следующем: все функции, используемые в наших моделях (например, `dot` или `+`), плавно и непрерывно преобразуют свои входные данные. Например, небольшое изменение  $u$  в операции  $z = x + u$  приведет к небольшому изменению  $z$  — и, зная направление изменения  $u$ , можно определить направление изменения  $z$ . Говоря математическим языком, данные функции *дифференцируемы*. Если объединить их в цепочку, получившаяся общая функция все равно будет дифференцируемой. Это утверждение, в частности, верно для функции, сопоставляющей веса модели с потерями в пакете данных. Небольшое изменение весов приводит к небольшому и предсказуемому изменению значения потерь, что позволяет использовать математический оператор, называемый *градиентом*, для описания изменения потерь при изменении весов модели в разных направлениях. Вычисленный градиент можно использовать для модификации весов (всех сразу в одном цикле, а не по одному) в направлении, уменьшающем потери.

Если вы уже знаете, что означает *дифференцируемость* и что такое *градиент*, можете сразу перейти к подразделу 2.4.3. Если нет — следующие два раздела помогут разобраться в данных понятиях.

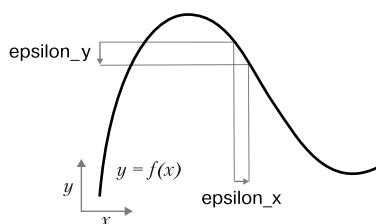
### 2.4.1. Что такое производная

Рассмотрим непрерывную гладкую функцию  $f(x) = y$ , отображающую число  $x$  в новое число  $y$ . Возьмем для примера функцию, изображенную на рис. 2.15.

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`, как показано на рис. 2.16.



**Рис. 2.15.** Непрерывная гладкая функция



**Рис. 2.16.** Для непрерывной функции небольшое изменение  $x$  даст в результате небольшое изменение  $y$

Кроме того, поскольку функция является *гладкой* (ее кривая не имеет острых углов), то при малых величинах `epsilon_x` в окрестностях определенной точки  $p$  функцию  $f$  можно аппроксимировать линейной функцией с наклоном  $a$ . Соответственно, `epsilon_y` можно вычислить как  $a * \text{epsilon}_x$ :

$$f(x + \text{epsilon}_x) = y + a * \text{epsilon}_x$$

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, небольшое изменение  $x$  в окрестностях  $p$  приведет к уменьшению  $f(x)$  (как показано на рис. 2.17); 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.

Для любой дифференцируемой функции  $f(x)$  (под *дифференцируемой* подразумевается «имеющей производную»: например, гладкие непрерывные функции могут иметь производную) существует такая производная функция  $f'(x)$ , которая отображает



**Рис. 2.17.** Производная  $f$  в точке  $p$

значения  $x$  в наклон локальной линейной аппроксимации  $f$  в этих точках. Например, производной от  $\cos(x)$  является  $-\sin(x)$ , производной от  $f(x) = a * x$  будет  $f'(x) = a$  и т. д.

Возможность получения производной функции — очень мощный инструмент, особенно для оптимизации задачи поиска значений  $x$ , минимизирующих значение  $f(x)$ . 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

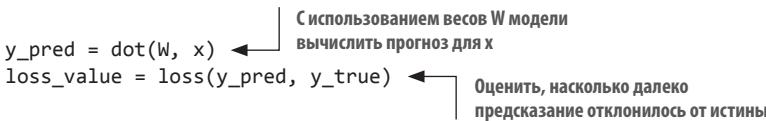
Функция, которую мы рассматривали выше, превращает скалярное значение  $x$  в другое скалярное значение  $y$ : ее можно изобразить в виде кривой на двумерной плоскости. Теперь представьте функцию, которая превращает кортеж скаляров  $(x, y)$  в скалярное значение  $z$ : это уже будет векторная операция. Ее можно изобразить как двумерную *поверхность* в трехмерном пространстве (с осями координат  $x, y, z$ ). Точно так же можно представить функции, принимающие на входе матрицы, трехмерные тензоры и т. д.

Понятие производной применимо к любой такой функции, если поверхности, которые они описывают, являются непрерывными и гладкими. Производная тензорной операции (или тензорной функции) называется *градиентом*. Градиенты — это просто обобщение понятия производной на функции, принимающие многомерные входные данные — тензоры. Надеюсь, вы помните, что производная скалярной функции выражает локальный наклон кривой функции? Таким же образом градиент тензорной функции выражает кривизну многомерной поверхности, описываемой функцией. Он характеризует изменение результата функции при изменении входных параметров.

Рассмотрим пример, базирующийся на машинном обучении. Дано:

- входной вектор  $x$  (образец в наборе данных);
- матрица  $W$  (веса модели);
- цель  $y_{\text{true}}$  (которую модель должна научиться ассоциировать с  $x$ );
- функция потерь `loss` (измеряет разницу между текущим прогнозом модели и  $y_{\text{true}}$ ).

С помощью  $W$  вычисляем приближение к цели  $y_{\text{pred}}$  и определяем потери или несоответствие между кандидатом  $y_{\text{pred}}$  и целью  $y_{\text{true}}$ :



Теперь, используя градиенты, можно выяснить, как обновить *W*, чтобы уменьшить значение *loss\_value*. Если входные данные *x* и *y\_true* зафиксированы, то предыдущие операции можно интерпретировать как функцию, отображающую значения *W* в значения потерь:

$$\text{loss\_value} = f(W) \leftarrow f \text{ описывает кривую (или многомерную поверхность),} \\ \text{образованную значениями потерь при изменении } W$$

Допустим, что *W0* — текущее значение *W*. Тогда производной функции *f* в точке *W0* будет тензор *grad(loss\_value, W0)* с той же формой, что и *W*, в котором каждый элемент *grad(loss\_value, W0)[i, j]* определяет направление и величину изменения в *loss\_value*, наблюдаемого при изменении *W0[i, j]*. Тензор *grad(loss\_value, W0)* — это градиент функции *f(W) = loss\_value* в точке *W0*, его также называют градиентом *loss\_value* для *W* в окрестностях *W0*.

### ЧАСТНЫЕ ПРОИЗВОДНЫЕ

Тензорную операцию *grad(f(W), W)* (которая принимает на входе матрицу *W*) можно выразить как комбинацию скалярных функций *grad\_ij(f(W), w\_ij)*, каждая из которых возвращает производную *loss\_value = f(W)* относительно веса *W[i, j]* в *W*, в предположении, что все остальные веса постоянны. *grad\_ij* называется *частной производной* функции *f* относительно *W[i, j]*.

Но что конкретно представляет собой *grad(loss\_value, W0)*? Выше вы видели, что производную функции *f(x)* единственного аргумента можно интерпретировать как наклон кривой *f*. Аналогично *grad(loss\_value, W0)* можно представить как тензор, описывающий *направление наискорейшего подъема* *loss\_value = f(W)* в окрестностях *W0* и наклон этого подъема. Каждая частная производная показывает наклон *f* в конкретном направлении.

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

### 2.4.3. Стохастический градиентный спуск

По идеи, 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. Добиться подобного можно, решив уравнение  $\text{grad}(f(W), W) = 0$  для  $W$ . Это полиномиальное уравнение с  $N$  переменными, где  $N$  — число весов в модели. Решить его для случая  $N = 2$  или  $N = 3$  не составляет труда, но для нейронных сетей, где число параметров редко бывает меньше нескольких тысяч, а часто достигает вообще нескольких десятков миллионов, — это практически неразрешимая задача.

Поэтому на практике используется алгоритм из четырех шагов, представленный в начале этого раздела. Параметры изменяются на небольшую величину, исходя из текущих значений потерь в случайному 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  $y_{\text{true}}$ .
2. Модель обрабатывает пакет  $x$  и получает пакет предсказаний  $y_{\text{pred}}$ .
3. Вычисляются потери модели на пакете, дающие оценку несовпадения между  $y_{\text{pred}}$  и  $y_{\text{true}}$ .
4. Вычисляется градиент потерь для весов модели (*обратный проход*).

Веса модели корректируются на небольшую величину в направлении, противоположном градиенту (например,  $W -= \text{скорость\_обучения} * \text{градиент}$ ), и тем самым снижаются потери. *Скорость обучения* — скалярный множитель, модулирующий «скорость» процесса градиентного спуска.

Выглядит довольно просто! Я только что описал *стохастический градиентный спуск на небольших пакетах* (mini-batch stochastic gradient descent, mini-batch SGD). Термин «*стохастический*» отражает тот факт, что каждый пакет данных выбирается случайно (в науке слово «*стохастический*» считается синонимом слова «*случайный*»). Рисунок 2.18 иллюстрирует происходящее на примере одномерных данных, когда модель имеет только один параметр и в вашем распоряжении есть только один обучающий образец.

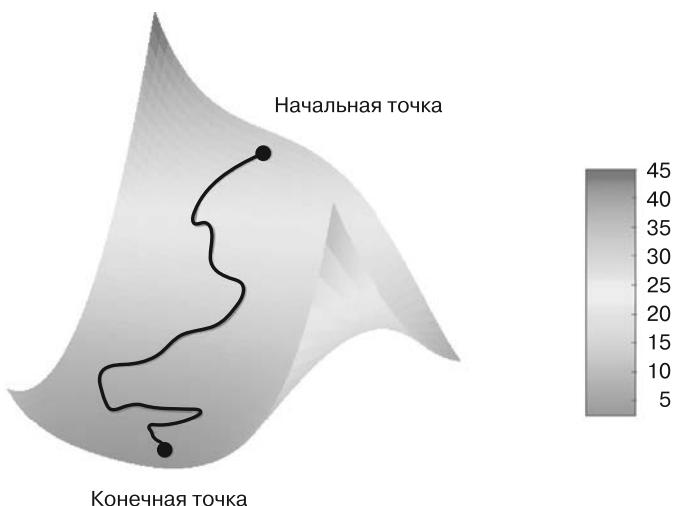


**Рис. 2.18.** Стохастический градиентный спуск вниз по одномерной кривой потерь (один обучаемый параметр)

Как можно заметить, выбор разумной величины скорости обучения имеет большое значение. Если взять ее слишком маленькой, спуск потребует большого количества итераций и может застрять в локальном минимуме. Если слишком большой – корректировки могут в конечном счете привести в абсолютно случайные точки на кривой.

Обратите внимание, что вариант алгоритма mini-batch SGD 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 градиентным спуском (batch gradient descent)*. Каждое изменение в этом случае будет более точным, но более затратным. Эффективным компромиссом между этими двумя крайностями является использование пакетов умеренного размера.

На рис. 2.18 изображен градиентный спуск в одномерном пространстве параметров, но на практике чаще используется градиентный спуск в пространствах с намного большим числом измерений: каждый весовой коэффициент в нейронной сети – это независимое измерение в пространстве, и их может быть десятки тысяч или даже миллионы. Чтобы лучше понять поверхности потерь, представьте градиентный спуск по двумерной поверхности, как показано на рис. 2.19. Но имейте в виду, что вам не удастся мысленно визуализировать фактический процесс обучения нейронной сети – с 1 000 000-мерным пространством этого не получится. Поэтому всегда помните, что представление, полученное на таких моделях с небольшим числом измерений, на практике может быть не всегда точным. В прошлом это часто приводило к ошибкам исследователей глубокого обучения.



**Рис. 2.19.** 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, внимание заслуживает идея *импульса*, которая используется во многих подобных вариантах. Импульс вводится для решения двух проблем SGD: невысокой скорости сходимости и попадания в локальный минимум. Взгляните на рис. 2.20, на котором изображена кривая потерь как функция параметра сети.

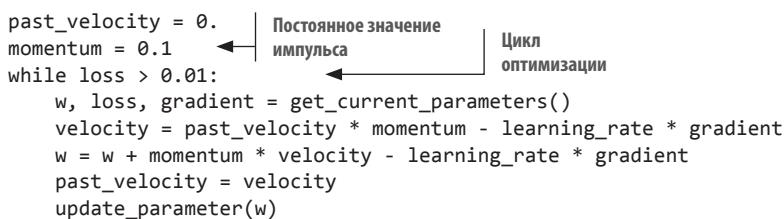


**Рис. 2.20.** Локальный и глобальный минимумы

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 маленькой скоростью обучения, 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, 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 и 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 и по величине предыдущего приращения параметра, как показано в следующей упрощенной реализации:

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    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. Но так ли это? Как на практике найти градиент сложных выражений? Как в двухслойной модели, с которой мы начали главу, получить градиент потерь с учетом весов? В этом нам поможет *алгоритм обратного распространения ошибки*.

##### Цепное правило

Обратное распространение — это способ использования производных простых операций (таких как сложение, `relu` или тензорное произведение) для упрощения вычисления градиента произвольно сложных комбинаций этих атомарных операций. Важно отметить, что нейронная сеть состоит из множества последовательных операций с тензорами, объединенных в одну цепочку,

каждая из которых имеет простую известную производную. Например, модель в листинге 2.2 можно выразить как функцию, параметризованную переменными  $w_1$ ,  $b_1$ ,  $w_2$  и  $b_2$  (принадлежащими первому и второму слоям `Dense` соответственно) и состоящую из атомарных операций `dot`, `relu`, `softmax` и `+`, а также функции потерь `loss`, которые легко дифференцируются:

```
loss_value = loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))
```

Согласно правилам дифференциального и интегрального исчисления такую цепочку функций можно вывести с помощью следующего тождества, называемого *правилом цепочки*.

Рассмотрим две функции —  $f$  и  $g$ , а также составную функцию  $fg$  такую, что  $fg(x) = f(g(x))$ :

```
def fg(x):
    x1 = g(x)
    y = f(x1)
    return y
```

Согласно цепному правилу  $\text{grad}(y, x) == \text{grad}(y, x_1) * \text{grad}(x_1, x)$ . Зная производные  $f$  и  $g$ , мы можем вычислить производную  $fg$ . Правило цепочки названо так потому, что при добавлении дополнительных промежуточных функций вычисления начинают выглядеть как цепочка:

```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y = f(x3)
    return y
grad(y, x) == (grad(y, x3) * grad(x3, x2) *
               grad(x2, x1) * grad(x1, x))
```

Применение цепного правила к вычислению значений градиента нейронной сети приводит к алгоритму, который называется *обратным распространением ошибки* (Backpropagation, *обратным дифференцированием*). Давайте посмотрим, как он работает.

## Автоматическое дифференцирование с графиками вычислений

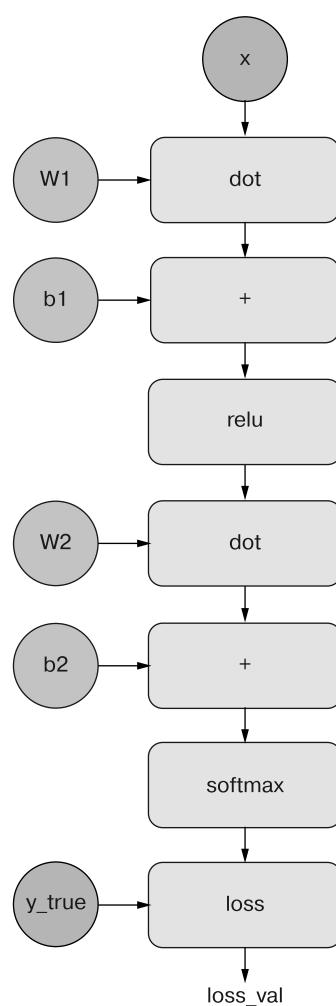
Обратное распространение можно представить в виде *графа вычислений*. Граф вычислений — это структура данных, лежащая в основе TensorFlow и давшая начало революции глубокого обучения в целом. Это ориентированный ациклический граф операций — в нашем случае тензорных. Например, взгляните на представление в виде графа вычислений нашей первой модели (рис. 2.21).

Графы вычислений оказались чрезвычайно успешной абстракцией в информатике, поскольку позволяют *рассматривать вычисления как данные*: последовательность вычислений кодируется как машиночитаемая структура данных, которую можно передать другой программе. Например, представьте программу, которая получает один граф вычислений и возвращает другой, новый, реализующий крупномасштабную распределенную версию того же вычисления, — подобное решение позволило бы превращать любые вычисления в распределенные без необходимости писать логику распределения самостоятельно. А что насчет программы, которая получает график вычислений и автоматически генерирует производную для выражения, которое представляет данный график? Сделать это намного проще, если вычисления выражены в виде явной структуры данных, а не, скажем, строк символов ASCII в файле `.py`.

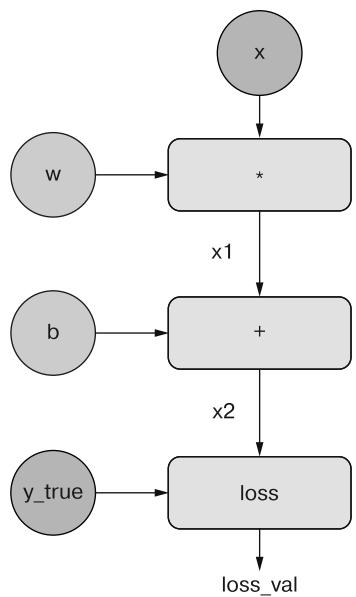
Для полноты картины рассмотрим несложный пример графа вычислений (рис. 2.22) — упрощенную версию графа, изображенного на рис. 2.21. Здесь у нас имеется только один линейный слой, и все переменные являются скалярными. Мы берем две скалярные переменные  $w$  и  $b$ , скалярный вход  $x$ , и применяем к ним некоторые операции, чтобы получить на выходе  $y$ . В заключение мы используем функцию вычисления абсолютных потерь:  $loss\_val = \text{abs}(y_{\text{true}} - y)$ . Поскольку нам нужно обновить  $w$  и  $b$  так, чтобы минимизировать  $loss\_val$ , мы должны вычислить  $\text{grad}(loss\_val, b)$  и  $\text{grad}(loss\_val, w)$ .

Давайте выберем конкретные значения для «входных узлов» в графике, то есть входные значения  $x$ ,  $y_{\text{true}}$ ,  $w$  и  $b$ , и распространим их через все узлы графа сверху вниз, пока не достигнем  $loss\_val$ . Это — *прямой проход* (рис. 2.23).

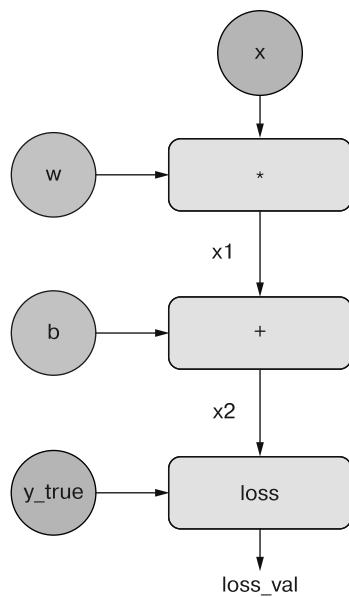
Затем «обратим» график: для каждого ребра в графике, идущего от  $A$  к  $B$ , создадим противоположное ребро от  $B$  к  $A$  и спросим, как сильно меняется  $B$  при изменении  $A$ . Иными словами, что такое  $\text{grad}(B, A)$ ? Подпишем каждое обратное ребро этим значением. Данный обратный график демонстрирует *обратный проход* (рис. 2.24).



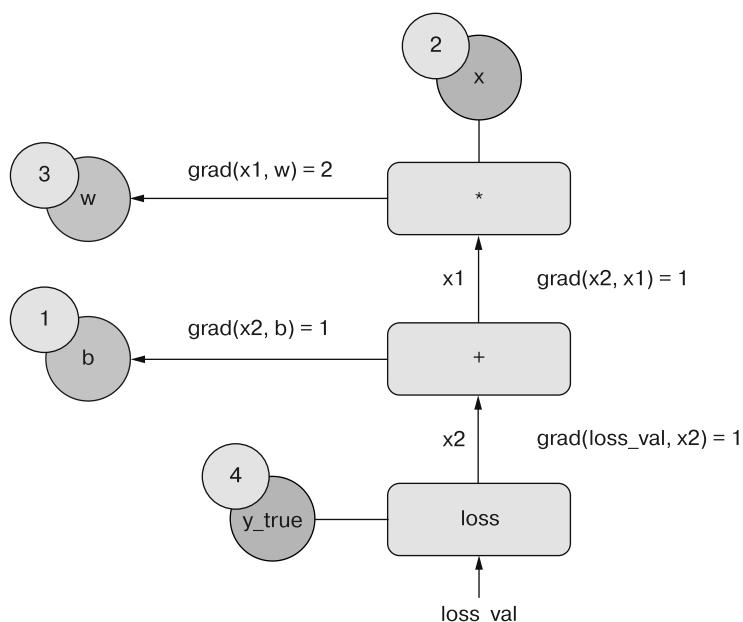
**Рис. 2.21.** Представление в виде графа вычислений нашей первой двухслойной модели



**Рис. 2.22.** Пример простого графа вычислений



**Рис. 2.23.** Прямой проход

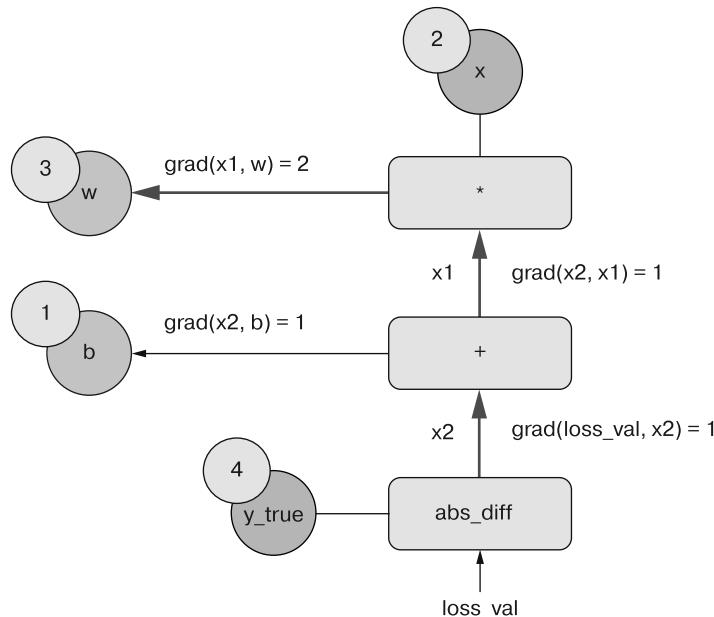


**Рис. 2.24.** Обратный проход

Мы имеем следующие результаты:

- $\text{grad}(\text{loss\_val}, \text{x2}) = 1$ , потому что с изменением  $\text{x2}$  на некоторую величину  $\text{loss\_val} = \text{abs}(4 - \text{x2})$  изменяется на ту же величину;
- $\text{grad}(\text{x2}, \text{x1}) = 1$ , потому что с изменением  $\text{x1}$  на некоторую величину  $\text{x2} = \text{x1} + \text{b} = \text{x1} + 1$  изменяется на ту же величину;
- $\text{grad}(\text{x2}, \text{b}) = 1$ , потому что с изменением  $\text{b}$  на некоторую величину  $\text{x2} = \text{x1} + \text{b} = 6 + \text{b}$  изменяется на ту же величину;
- $\text{grad}(\text{x1}, \text{w}) = 2$ , потому что с изменением  $\text{w}$  на некоторую величину  $\text{x1} = \text{x} * \text{w} = 2 * \text{w}$  изменяется на величину, в два раза большую.

Применив цепное правило к обратному графу, можно получить производную узла по отношению к другому узлу, *перемножив производные всех ребер на пути, соединяющем два узла*. Например,  $\text{grad}(\text{loss\_val}, \text{w}) = \text{grad}(\text{loss\_val}, \text{x2}) * \text{grad}(\text{x2}, \text{x1}) * \text{grad}(\text{x1}, \text{w})$  (рис. 2.25).



**Рис. 2.25.** Путь от  $\text{loss\_val}$  до  $w$  в обратном графе

Применение цепного правила к нашему графу дает нам искомое:

- $\text{grad}(\text{loss\_val}, \text{w}) = 1 * 1 * 2 = 2$ ;
- $\text{grad}(\text{loss\_val}, \text{b}) = 1 * 1 = 1$ .

### ПРИМЕЧАНИЕ

Если в обратном графе есть несколько путей, связывающих два узла, а и b, то получить  $\text{grad}(b, a)$  можно суммированием вкладов всех путей.

Вы только что увидели обратное распространение в действии! Обратное распространение – это просто применение цепного правила к графу вычислений и ничего более. Оно начинается с конечного значения потери и движется в обратном направлении, от верхних слоев к нижним, используя цепное правило для вычисления вклада каждого параметра в значение потери. Отсюда и название «обратное распространение»: мы «распространяем в обратном направлении» вклады в потери различных узлов в графе.

В настоящее время нейронные сети конструируются с использованием современных фреймворков, поддерживающих *автоматическое дифференцирование* (таких как TensorFlow). Автоматическое дифференцирование основано на применении графов вычислений, подобных тем, что вы видели выше, и позволяет извлекать градиенты произвольных последовательностей дифференцируемых тензорных операций, не выполняя при этом никакой дополнительной работы, кроме записи прямого прохода. Когда я создавал свои первые нейронные сети на C в 2000-х, мне приходилось писать реализацию градиентов вручную. Теперь благодаря современным инструментам автоматического дифференцирования нет необходимости самостоятельно реализовывать обратное распространение. Считайте, что вам повезло!

### Объект GradientTape в TensorFlow

Роль интерфейса для управления мощными возможностями автоматического дифференцирования в TensorFlow играет `GradientTape`. Это объект на Python, который «записывает» выполняемые тензорные операции в форме графа вычислений (иногда называемого tape – лентой). Этот график затем можно использовать для получения градиента любого результата относительно любой переменной или набора переменных (экземпляров класса `tf.Variable`). Класс `tf.Variable` представляет особый вид тензора, предназначенный для хранения изменяемого состояния: например, веса нейронной сети всегда являются экземплярами `tf.Variable`.

```
import tensorflow as tf
x = tf.Variable(0.)           | Создать экземпляр Variable
with tf.GradientTape() as tape: | Открыть контекст
    y = 2 * x + 3            | Применить некоторые тензорные операции
grad_of_y_wrt_x = tape.gradient(y, x) | к нашей переменной внутри контекста
                                         | Использовать экземпляр tape
                                         | для извлечения градиента выходного
                                         | значения у относительно переменной x
```

`GradientTape` работает с тензорными операциями:

```
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Создать экземпляр `Variable`  
с формой  $(2, 2)$  и с нулевыми  
начальными значениями элементов

`grad_of_y_wrt_x` — тензор с формой  $(2, 2)$   
как  $x$ , описывающий кривизну  $y = 2 * a + 3$   
в окрестностях  $x = [[0, 0], [0, 0]]$

И со списками переменных:

```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

`matmul` — так называется скалярное  
произведение в TensorFlow

`grad_of_y_wrt_W_and_b` — это  
список с двумя тензорами, формы  
которых совпадают с формами  $W$  и  $b$   
соответственно

В следующей главе вы поближе познакомитесь с `GradientTape`.

## 2.5. ОГЛЯДЫВАЯСЬ НА ПЕРВЫЙ ПРИМЕР

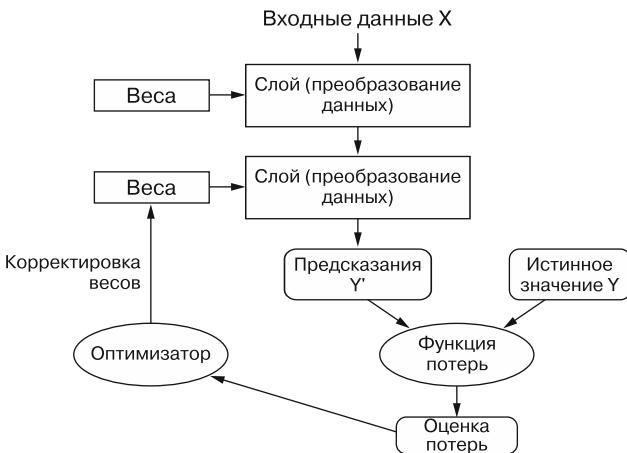
Мы подошли к концу главы, и теперь вы должны неплохо представлять, что происходит в недрах нейронной сети. То, что в начале главы казалось волшебным черным ящиком, сложилось в более ясную картину, изображенную на рис. 2.26. Итак, у нас есть модель, состоящая из слоев, которая преобразует входные данные в прогнозы. Затем функция потерь сравнивает прогнозы с целевыми значениями, получая значение потерь: меру соответствия полученного моделью прогноза ожидаемому результату. Позже оптимизатор использует значение потерь для корректировки весов модели.

Давайте вернемся назад, к первому примеру, и рассмотрим каждую его часть через призму полученных вами знаний.

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 вам известно, что входные изображения хранятся в тензорах NumPy типа `float32`, имеющих форму  $(60000, 784)$  (обучающие данные) и  $(10000, 784)$  (контрольные данные) соответственно.



**Рис. 2.26.** Связь между слоями, функцией потерь и оптимизатором в сети

Вот наша сеть:

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Вы уже знаете, что эта модель состоит из цепочки двух слоев `Dense`, каждый из которых применяет к входным данным несколько простых тензорных операций, а также что эти операции вовлекают весовые тензоры. Весовые тензоры, являющиеся атрибутами слоев, — это место, где запоминаются *знания*, накопленные моделью.

Вот как выглядел этап компиляции:

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Сейчас вы понимаете, что `sparse_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:

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

Теперь вам понятно, что происходит в вызове `fit`: модель начинает перебирать обучающие данные мини-пакетами по 128 образцов и выполняет пять итераций (каждая итерация по всем обучающим данным называется *эпохой*). Для каждого мини-пакета модель вычисляет градиенты потерь относительно весов (с использованием алгоритма обратного распространения ошибки, который использует цепное правило дифференциального и интегрального исчисления) и изменяет значения весов в соответствующем направлении.

В течение пяти эпох сеть выполнит 2345 изменений градиента (по 469 на эпоху), после чего потери модели окажутся достаточно низкими, чтобы она могла классифицировать рукописные цифры с высокой точностью.

Итак, вы знаете большую часть того, что нужно знать о нейронных сетях. Давайте подтвердим это, повторно реализовав в TensorFlow упрощенную версию нашего первого примера.

### 2.5.1. Повторная реализация первого примера в TensorFlow

Что лучше покажет полное и однозначное понимание темы, чем реализация с нуля? Конечно, понятие «с нуля» здесь довольно относительное: мы не будем повторно писать базовые тензорные операции и реализацию обратного распространения. Но мы опустимся на такой низкий уровень, что нам практически не понадобятся функции из библиотеки Keras.

Не волнуйтесь, если что-то в примере останется непонятым. В следующей главе мы подробно рассмотрим TensorFlow API. А пока просто попытайтесь ухватить суть происходящего. Цель этого примера — помочь кристаллизовать понимание математики глубокого обучения с использованием конкретной реализации. Поехали!

#### Простой класс `Dense`

Ранее вы узнали, что слой `Dense` реализует следующее преобразование входных данных, где `W` и `b` — параметры модели, а `activation` — поэлементная функция (обычно `relu`, но в последнем слое — `softmax`):

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

Реализуем на Python простой класс `NaiveDense`, создающий две переменные TensorFlow, `W` и `b`, и имеющий метод `__call__()`, который применяет предыдущее преобразование.

```

import tensorflow as tf

class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        self.activation = activation

        w_shape = (input_size, output_size)
        w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
        self.W = tf.Variable(w_initial_value)

        b_shape = (output_size,)
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)

    def __call__(self, inputs):
        return self.activation(tf.matmul(inputs, self.W) + self.b)

    @property
    def weights(self):
        return [self.W, self.b]

```

## Простой класс Sequential

Теперь создадим класс `NaiveSequential`, объединяющий слои в цепочку. Он обертывает список слоев и реализует метод `__call__()`, который просто вызывает по порядку слои в этом списке, передавая входные данные. Он также имеет свойство `weights`, упрощающее наблюдение за весами слоев.

```

class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
            x = layer(x)
        return x

    @property
    def weights(self):
        weights = []
        for layer in self.layers:
            weights += layer.weights
        return weights

```

Используя классы `NaiveDense` и `NaiveSequential`, можно создать имитацию модели Keras:

```

model = NaiveSequential([
    NaiveDense(input_size=28 * 28, output_size=512, activation=tf.nn.relu),
    NaiveDense(input_size=512, output_size=10, activation=tf.nn.softmax)
])
assert len(model.weights) == 4

```

## Генератор пакетов

Нам также нужен механизм, который поможет выполнять итерации по пакетам и изображениям из набора MNIST. Реализуется такой механизм просто:

```
import math

class BatchGenerator:
    def __init__(self, images, labels, batch_size=128):
        assert len(images) == len(labels)
        self.index = 0
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.num_batches = math.ceil(len(images) / batch_size)

    def next(self):
        images = self.images[self.index : self.index + self.batch_size]
        labels = self.labels[self.index : self.index + self.batch_size]
        self.index += self.batch_size
        return images, labels
```

## 2.5.2. Выполнение одного этапа обучения

Этап обучения — самая сложная часть процесса. Нам требуется скорректировать веса модели после обучения на одном пакете данных. Для этого нужно сделать следующее.

1. Вычислить прогнозы для изображений в пакете.
2. Найти значения потерь для этих прогнозов с учетом фактических меток.
3. Вычислить градиент потерь с учетом весов модели.
4. Скорректировать веса на небольшую величину в направлении, противоположном градиенту.

Для вычисления градиента используем объект `GradientTape` из библиотеки TensorFlow, который был представлен в пункте 2.4.4:

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        per_sample_losses = tf.keras.losses.sparse_categorical_crossentropy(
            labels_batch, predictions)
        average_loss = tf.reduce_mean(per_sample_losses)
    gradients = tape.gradient(average_loss, model.weights) ←
    update_weights(gradients, model.weights)
    return average_loss
```

**Скорректировать веса с учетом градиентов (эту функцию мы определим ниже)**

**Вычислить градиент потерь с учетом весов. Результат `gradients` — это список, каждый элемент которого соответствует весу в списке `model.weights`**

**Выполнить «прямой проход» (вычислить прогноз модели в контексте `GradientTape`)**

Как вы уже знаете, цель шага «корректировки весов» (представленного в предыдущем листинге функцией `update_weights`) состоит в том, чтобы «чуть-чуть» скорректировать веса в направлении, которое уменьшит потери в этом пакете. Величина корректировки определяется «скоростью обучения», обычно небольшой. Самый простой способ реализовать функцию `update_weights` — вычесть `gradient * learning_rate` из каждого веса:

```
learning_rate = 1e-3

def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign_sub(g * learning_rate) ← assign_sub — это эквивалент
                                                оператора -= для переменных
                                                TensorFlow
```

На практике вам редко придется задумываться о реализации корректировки вручную, потому что обычно для этого используется экземпляр оптимизатора из Keras, например:

```
from tensorflow.keras import optimizers

optimizer = optimizers.SGD(learning_rate=1e-3)

def update_weights(gradients, weights):
    optimizer.apply_gradients(zip(gradients, weights))
```

Теперь, покончив с этапом обучения на одном пакете, перейдем к реализации целой эпохи обучения.

### 2.5.3. Полный цикл обучения

Эпоха обучения просто повторяет шаг обучения для каждого пакета обучающих данных, а полный цикл обучения — это повторение одной эпохи:

```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")
        batch_generator = BatchGenerator(images, labels)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            if batch_counter % 100 == 0:
                print(f"loss at batch {batch_counter}: {loss:.2f}")
```

Давайте протестируем получившееся:

```
from tensorflow.keras.datasets import mnist
(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

fit(model, train_images, train_labels, epochs=10, batch_size=128)
```

## 2.5.4. Оценка модели

Мы можем оценить модель, применив `argmax` к прогнозам на контрольных изображениях и сравнив с ожидаемыми метками:

```
predictions = model(test_images)
predictions = predictions.numpy() ← Метод .numpy() преобразует
predicted_labels = np.argmax(predictions, axis=1) тензор TensorFlow в тензор NumPy
matches = predicted_labels == test_labels
print(f"accuracy: {matches.mean():.2f}")
```

Вот и все! Как видите, довольно сложно реализовать «вручную» то, что можно выполнить с помощью нескольких строк кода, использующих Keras. Но теперь, пройдя через эти этапы, вы должны четко уяснить происходящее внутри нейронной сети при вызове ее метода `fit()`. Понимание низкоуровневых деталей поможет вам эффективнее использовать высокуюровневые функции Keras API.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Тензоры образуют основу современных систем машинного обучения. Они бывают разных видов в зависимости от типа (`dtype`), ранга (`rank`) и формы (`shape`).
- Числовыми тензорами можно манипулировать с помощью *тензорных операций* (таких как сложение, тензорное произведение или поэлементное умножение), каждая из которых имеет геометрическую интерпретацию. Вообще, все в глубоком обучении имеет геометрическую интерпретацию.
- Модели глубокого обучения состоят из цепочек простых тензорных операций, параметризованных *весами*, которые сами являются тензорами. Веса модели — это место, где хранятся накопленные «знания».
- *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 модели с учетом потерь в пакете. Затем параметры модели немного смещаются (величина смещения определяется скоростью обучения) в направлении, противоположном направлению градиента. Это называется *стохастическим градиентным спуском на небольших пакетах*.

- Процесс обучения становится возможным благодаря тому, что все тензорные операции в нейронных сетях являются дифференцируемыми и, следовательно, позволяют применять цепное правило для вывода функции градиента, отображающей текущие параметры и текущий пакет данных в значение градиента. Это называется *обратным распространением ошибки*.
- В последующих главах вам часто будут встречаться два ключевых понятия — *функции потерь* и *оптимизаторы*. Они должны быть определены до передачи данных в модель.
  - *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.

# *Введение в Keras и TensorFlow*

---

## **В этой главе**

- ✓ Библиотеки TensorFlow и Keras и взаимоотношения между ними.
- ✓ Настройка окружения для глубокого обучения.
- ✓ Реализация базовых концепций глубокого обучения в Keras и TensorFlow.

Цель этой главы — дать все необходимое, чтобы вы могли начать применять глубокое обучение на практике. Я кратко расскажу о Keras (<https://keras.io>) и TensorFlow (<https://tensorflow.org>) — инструментах глубокого обучения на языке Python, которые мы будем использовать на протяжении всей книги. Вы узнаете, как настроить окружение для глубокого обучения с TensorFlow, Keras, а также с поддержкой вычислений на графическом процессоре. Наконец, опираясь на знания, полученные во время нашего первого знакомства с Keras и TensorFlow в главе 2, мы рассмотрим основные компоненты нейронных сетей и то, как они преобразуются в код, использующий Keras и TensorFlow.

К концу этой главы вы будете готовы перейти к решению практических задач — чем мы и займемся в главе 4.

### 3.1. ЧТО ТАКОЕ TENSORFLOW

TensorFlow – бесплатная платформа машинного обучения на Python с открытым исходным кодом, разработанная в основном в Google. Как и у NumPy, основная цель TensorFlow – дать инженерам и исследователям возможность манипулировать математическими выражениями с числовыми тензорами. Но TensorFlow может намного больше, чем NumPy, в том числе:

- автоматически вычислять градиент любого дифференцируемого выражения (как было показано в главе 2), что делает ее прекрасной основой для машинного обучения;
- работать не только на обычных, но также на графических и тензорных процессорах – высокопараллельных аппаратных ускорителях;
- распределять вычисления между множеством компьютеров;
- экспортirовать вычисления другим окружениям выполнения, таким как C++, JavaScript (для веб-приложений, выполняющихся в браузере) или TensorFlow Lite (для приложений, действующих в мобильных или встраиваемых устройствах) и т. д. Это упрощает развертывание приложений TensorFlow в практических условиях.

Важно помнить, что TensorFlow – это не просто библиотека. Это целая платформа, на которой базируется обширная экосистема компонентов. Часть из них разработана в Google, часть – сторонними организациями. В их числе можно назвать TF-Agents для исследования обучения с подкреплением, TFX для организации управления процессом машинного обучения, TensorFlow Serving для развертывания в производственном окружении и репозиторий TensorFlow Hub предварительно обученных моделей. Вместе эти компоненты охватывают широкий спектр сценариев использования, от передовых исследований до крупномасштабных производственных приложений.

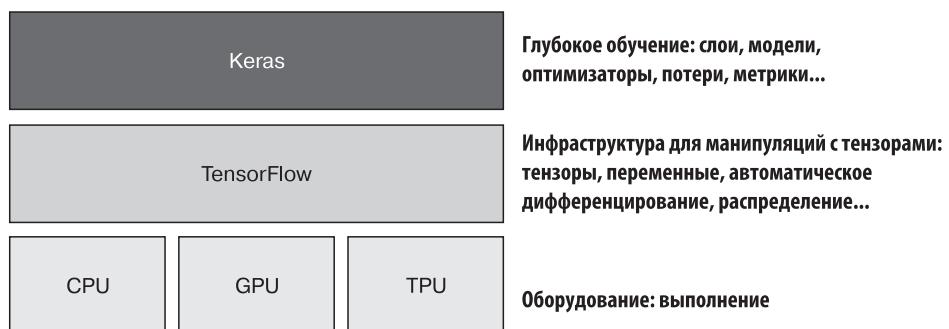
Библиотека TensorFlow поддерживает возможность масштабирования: так, ученые из Национальной лаборатории Ок-Риджа с ее помощью обучили модель прогнозирования экстремальных погодных условий на суперкомпьютере IBM Summit с 27 000 графических процессоров и суммарной производительностью 1,1 экафлопса. Google тоже использовал TensorFlow для разработки приложений глубокого обучения, требующих больших вычислительных ресурсов, таких как агент AlphaZero для игры в шахматы и в го. При создании своих моделей, если вам будет выделен достаточный бюджет, вы можете надеяться на масштабирование до 10 петафлопс на небольшом модуле TPU или большом кластере графических процессоров, арендованном в Google Cloud или AWS. Это составит около 1 % от пиковой производительности одного из самых мощных суперкомпьютеров в 2019 году!

## 3.2. ЧТО ТАКОЕ KERAS

Keras — это библиотека глубокого обучения для Python, основанная на TensorFlow, которая обеспечивает удобный способ определения и тренировки моделей глубокого обучения. Первоначально Keras создавалась для исследований с целью упростить эксперименты с глубоким обучением.

Благодаря TensorFlow библиотека Keras может работать на различных типах оборудования (рис. 3.1) — графическом, тензорном или обычном процессоре — и поддерживает простую возможность распределения вычислений между тысячами компьютеров.

Особое внимание в Keras уделяется опыту разработчиков. Данная библиотека предназначена для людей, а не для машин. Она следует передовым методам снижения когнитивной нагрузки, предлагая простые и логичные рабочие процессы, минимизируя количество действий, необходимых для типичных случаев использования, и давая четкую и действенную обратную связь на ошибки пользователя. Это упрощает освоение Keras начинающими исследователями и увеличивает продуктивность экспертов.



**Рис. 3.1.** Keras и TensorFlow: TensorFlow — это низкоуровневая платформа тензорных вычислений, а Keras — высокоуровневая библиотека глубокого обучения

По состоянию на конец 2021 года насчитывалось более одного миллиона пользователей Keras: от исследователей, инженеров и специалистов по обработке данных в начинающих и крупных компаниях до аспирантов и любителей. Keras используется в Google, Netflix, Uber, CERN, NASA, Yelp, Instacart, Square и сотнях стартапов, работающих над решением широкого круга задач во всех отраслях. Рекомендации на YouTube для вас подбираются моделями Keras. Беспилотные автомобили Waymo также управляются ими. Keras популярна на

Kaggle — веб-сайте, проводящем соревнования по машинному обучению, большинство из которых было выиграно с использованием данного фреймворка.

Такое число пользователей библиотека Keras приобрела потому, что не вынуждает следовать одному «истинному» способу конструирования и обучения моделей. Вместо этого она дает возможность применять широкий спектр подходов, соответствующих уровню подготовки пользователей, от очень высокого до очень низкого. Например, в вашем распоряжении множество способов конструирования моделей и множество способов их обучения, каждый из которых предлагает определенные компромиссы между удобством и гибкостью. В главе 5 мы подробно рассмотрим значительную их часть. Вы можете использовать Keras так же, как использовали бы Scikit-learn — просто вызывая `fit()` и позволяя фреймворку выполнить свою работу, — или как NumPy, определяя и управляя всеми самыми мелкими деталями.

А значит, все знания, приобретаемые вами в самом начале пути, сохранят актуальность, когда вы станете экспертом. Keras поможет быстро начать работу, а затем постепенно погружаться в рабочие процессы и писать с нуля все больше и больше логики. Вам не придется переключаться на совершенно другой фреймворк, когда вы вырастете от студента до исследователя или от специалиста по данным до инженера глубокого обучения.

Эта философия мало чем отличается от философии самого Python! Некоторые языки (например, объектно-ориентированные или функциональные) дают только один способ написания программ. Python — многопарадигменный язык: он предлагает множество возможных подходов к программированию, прекрасно уживающихся вместе. Это делает Python пригодным для использования в самых разных случаях: для системного администрирования, анализа данных, машинного обучения, веб-разработки... или просто для обучения программированию. Точно так же Keras можно рассматривать как диалект Python для глубокого обучения: удобный язык глубокого обучения, предлагающий множество подходов для пользователей с разным уровнем подготовки.

### 3.3. KERAS И TENSORFLOW: КРАТКАЯ ИСТОРИЯ

Среда Keras старше TensorFlow на восемь месяцев. Она была выпущена в марте 2015 года, а TensorFlow — в ноябре. Вы можете спросить: если Keras основана на TensorFlow, как она могла появиться раньше? Дело в том, что первоначально Keras основывалась на Theano — еще одной библиотеке для работы с тензорами, обеспечивавшей автоматическое дифференцирование и поддержку вычислений на графических процессорах, самой первой в своем классе. Theano была разработана в Монреальском институте алгоритмов обучения (Montréal Institute

for Learning Algorithms, MILA) при Монреальском университете и во многих отношениях может считаться предшественницей TensorFlow. В ней впервые была реализована идея использования статических графов вычислений для автоматического дифференцирования и компиляции кода для выполнения на CPU и GPU.

В конце 2015 года, после выпуска TensorFlow, архитектура Keras была преобразована для поддержки нескольких базовых библиотек: появилась возможность выбора между Theano и TensorFlow, при этом переключение было таким же простым, как изменение переменной окружения. К сентябрю 2016 года TensorFlow достигла достаточно высокого уровня технической зрелости, чтобы использовать ее в качестве опции по умолчанию. В 2017 году в Keras была добавлена поддержка еще двух библиотек тензорных операций: CNTK (разработана в Microsoft) и MXNet (в Amazon). В настоящее время разработка Theano и CNTK прекратилась, а MXNet не получила широкого распространения за пределами Amazon. Keras снова стала библиотекой, основанной на одном тензорном фреймворке — TensorFlow.

Keras и TensorFlow уже много лет успешно сосуществуют вместе. В течение 2016 и 2017 годов Keras приобрела широкую известность как удобное средство для разработки приложений TensorFlow, привлекающее новых пользователей в экосистему TensorFlow. К концу 2017 года большинство пользователей фреймворка TensorFlow использовали его через Keras или в сочетании с Keras. В 2018 году руководство TensorFlow выбрало Keras в качестве официального высокоуровневого интерфейса TensorFlow. В результате библиотека Keras заняла центральное место в версии TensorFlow 2.0, выпущенной в сентябре 2019 года, — кардинально переделанного комплекса TensorFlow и Keras, учитывающего отзывы пользователей и технический прогресс за предыдущие четыре года.

Теперь вы готовы начать использовать код для Keras и TensorFlow на практике. Приступим.

## **3.4. НАСТРОЙКА ОКРУЖЕНИЯ ДЛЯ ГЛУБОКОГО ОБУЧЕНИЯ**

Прежде чем приступить к разработке приложений глубокого обучения, нужно настроить рабочее окружение. Для выполнения кода, реализующего глубокое обучение, рекомендуется (но это не обязательно) использовать современный графический процессор NVIDIA. Некоторые приложения — в частности, для обработки изображений с применением сверточных сетей — показывают крайне низкую производительность даже на очень быстрых многоядерных CPU. И даже

приложениям, которые вполне могут выполняться на CPU, выполнение на современном GPU часто дает прирост скорости примерно в 5–10 раз.

Есть три варианта настройки окружения для глубокого обучения на графическом процессоре:

- купить и установить на рабочую станцию физический графический процессор NVIDIA;
- использовать экземпляры GPU в Google Cloud или AWS EC2;
- использовать бесплатную среду выполнения на графическом процессоре от Colaboratory — службы для блокнотов Jupyter, поддерживаемой компанией Google (мы рассмотрим эти блокноты подробнее в следующем разделе).

Служба Colaboratory предлагает самый простой способ начать работу: она не требует покупки оборудования и установки программного обеспечения — просто откройте вкладку в браузере и приступайте к программированию. Именно этот вариант я рекомендую для выполнения примеров этой книги. Однако бесплатная версия Colaboratory подходит только для небольших рабочих нагрузок. Для масштабных проектов вам придется использовать первый или второй вариант.

Если у вас еще нет GPU (последней, высокопроизводительной модели NVIDIA GPU), который можно было бы использовать для нужд глубокого обучения, 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.

Однако тем, кто планирует заниматься глубоким обучением всерьез, такой подход не годится — он не подойдет даже новичкам, собирающимся фокусироваться на теме дольше нескольких месяцев. Облачные экземпляры недешевы: один час работы графического процессора V100 в Google Cloud стоил 2,48 доллара. Между тем хороший графический процессор потребительского класса обойдется вам от 1500 до 2500 долларов. 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.

Кроме того, независимо от окружения, локального или облачного, лучше взять рабочую станцию Unix. Технически библиотеку Keras можно использовать непосредственно в Windows, но я не рекомендую этого. Если у вас Windows и вы хотите заниматься глубоким обучением на собственной рабочей станции, самое простое решение — установить Ubuntu второй операционной системой или использовать подсистему Windows Subsystem for Linux (WSL) — слой совместимости, позволяющий запускать приложения для Linux в Windows.

Может показаться, что это слишком хлопотно, однако подобный подход поможет сэкономить вам массу времени и избавит от многих проблем в будущем.

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

Блокноты Jupyter Notebook — отличный способ проведения экспериментов по глубокому обучению и, в частности, апробации примеров этой книги. Они широко применяются в сообществах машинного обучения и науки о данных. *Блокнот* (notebook) — 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 богатым оформлением. Блокноты позволяют разбить объемный эксперимент на несколько коротких шагов, выполняемых 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.

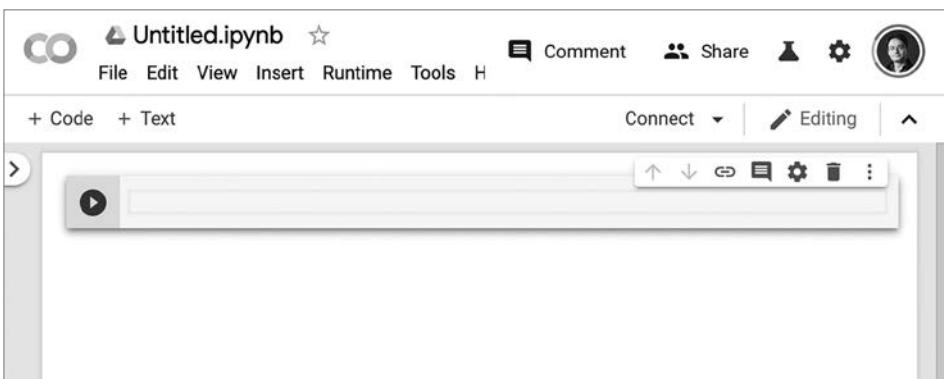
Я настоятельно рекомендую использовать блокноты Jupyter Notebook на первых порах работы с Keras, хотя это и не является обязательным требованием: вы можете также запускать автономные сценарии на Python или выполнять код в интегрированной среде, такой как PyCharm. Все примеры данной книги доступны в виде блокнотов Jupyter с открытым исходным кодом на сайте GitHub: [github.com/fchollet/deep-learning-with-python-notebooks](https://github.com/fchollet/deep-learning-with-python-notebooks).

### **3.4.2. Использование Colaboratory**

Colaboratory (или просто Colab) — это бесплатная облачная служба для блокнотов Jupyter, не требующая установки дополнительного программного обеспечения. По сути, это веб-страница, позволяющая сразу же писать и выполнять сценарии, использующие Keras. Она дает доступ к бесплатной (но ограниченной) среде выполнения на графическом процессоре и даже к среде выполнения на тензорном процессоре (TPU), благодаря чему вам не придется покупать свой GPU. Рекомендую использовать Colaboratory для выполнения примеров данной книги.

#### **Первые шаги с Colaboratory**

Для начала работы с Colab откройте страницу <https://colab.research.google.com> и нажмите кнопку New Notebook (Создать блокнот). Вы увидите стандартный интерфейс блокнота, показанный на рис. 3.2.

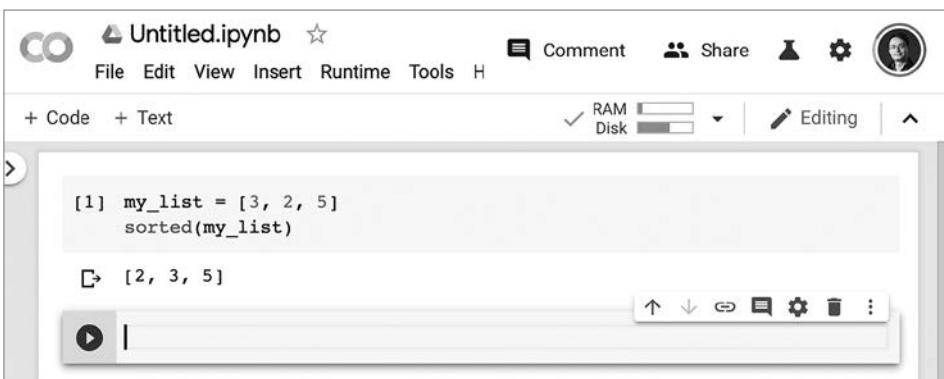


**Рис. 3.2.** Новый блокнот в Colab

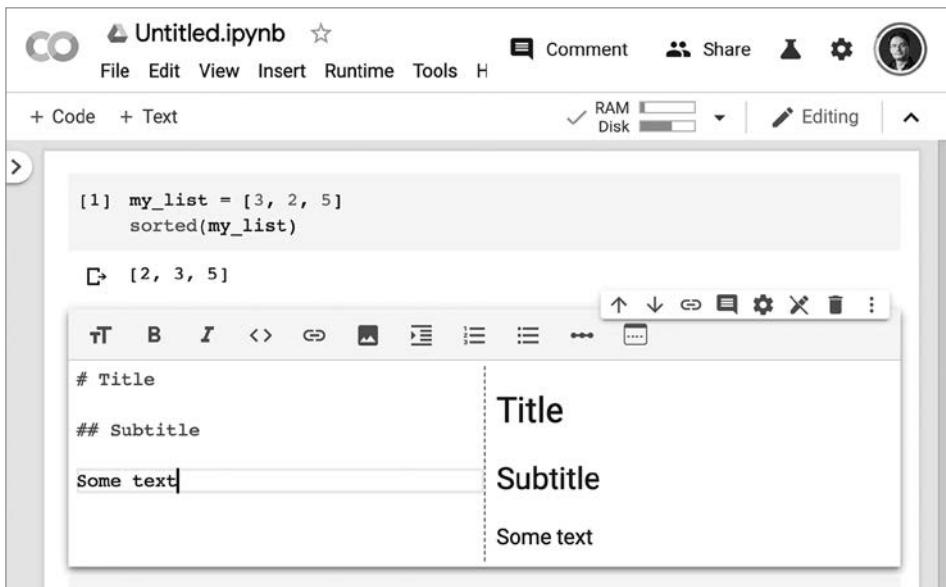
Обратите внимание на две кнопки на панели инструментов: + Code (+ Код) и + Text (+ Текст). Они предназначены для создания ячеек с выполняемым кодом на Python и с текстовыми комментариями соответственно. После ввода кода в нужную ячейку нажмите Shift+Enter, чтобы выполнить его (рис. 3.3).

В текстовой ячейке можете использовать синтаксис языка разметки Markdown (рис. 3.4). Точно так же, закончив ввод текста, нажмите Shift+Enter, чтобы отобразить его.

Текстовые ячейки помогают сделать структуру блокнотов удобочитаемой: их можно использовать для описания кода, добавляя подзаголовки и абзацы с пояснениями, а также рисунки. Следовательно, опыт использования блокнотов должен быть интерактивным!



**Рис. 3.3.** Создание ячейки с выполняемым кодом



**Рис. 3.4.** Создание текстовой ячейки

### Установка пакетов с помощью pip

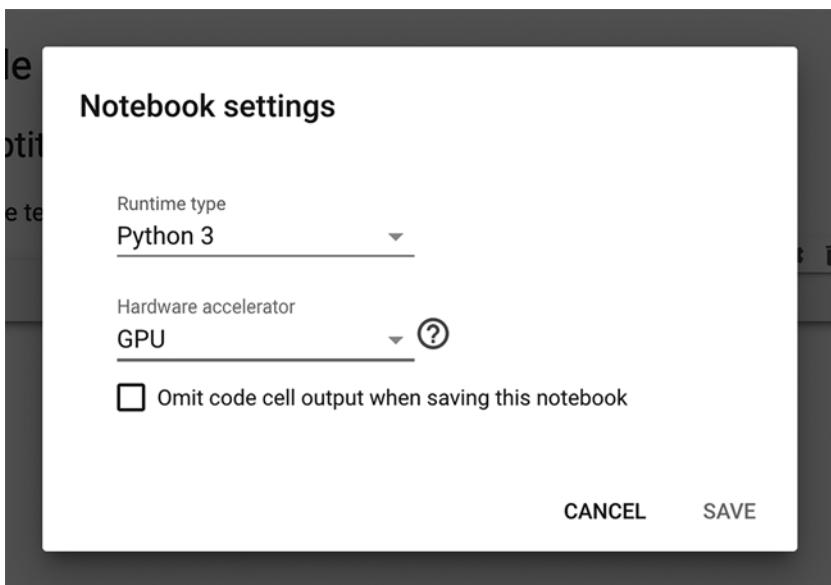
По умолчанию среда Colab уже включает библиотеки TensorFlow и Keras, поэтому можно сразу начинать использовать ее без необходимости выполнять какие-то действия по установке. Но если вдруг понадобится установить дополнительный пакет с помощью `pip`, это легко сделать, использовав следующий синтаксис в ячейке для кода (обратите внимание, что строка начинается с восклицательного знака (!), чтобы показать, что это команда оболочки, а не код на Python):

```
!pip install package_name
```

### Работа со средой выполнения GPU

Чтобы начать работу со средой выполнения GPU в Colab, выберите в меню пункт Runtime ▶ Change Runtime Type (Среда выполнения ▶ Сменить среду выполнения) и в раскрывающемся списке Hardware Accelerator (Аппаратный ускоритель) выберите GPU (рис. 3.5).

Библиотеки TensorFlow и Keras автоматически используют графический процессор, если он доступен, поэтому после выбора среды выполнения GPU вам не придется делать ничего.



**Рис. 3.5.** Выбор среды выполнения GPU в Colab

Обратите внимание, что в раскрывающемся списке Hardware Accelerator (Аппаратный ускоритель) присутствует также пункт TPU. В отличие от среды выполнения GPU, для использования TPU в TensorFlow и Keras вам потребуется выполнить в своем коде небольшую ручную настройку. Я расскажу об этом в главе 13, а до тех пор для примеров данной книги советую брать среду выполнения GPU.

Теперь, организовав окружение для запуска кода Keras, посмотрим, как основные идеи, с которыми мы познакомились в главе 2, переводятся на язык Keras и TensorFlow.

## 3.5. ПЕРВЫЕ ШАГИ С TENSORFLOW

Как мы уже знаем из предыдущих глав, обучение нейронных сетей основывается на следующих концепциях:

- во-первых, тензоры и низкоуровневые операции с ними — основа всего современного машинного обучения. В TensorFlow это:
  - *тензоры*, в том числе специальные, хранящие состояние сети (*переменные*);
  - *тензорные операции*, такие как сложение, `relu`, `matmul`;

- *обратное распространение* — механизм вычисления градиента математических выражений (в TensorFlow этот механизм предоставляет объект `GradientTape`);
- во-вторых, высокоуровневые идеи глубокого обучения. В Keras это:
  - *слои*, которые объединяются в модель;
  - *функция потерь*, которая определяет сигнал обратной связи, используемый для обучения;
  - *оптимизатор*, определяющий порядок продвижения обучения;
  - *метрики* для оценки качества модели (такие как точность);
  - *цикл обучения*, действующий методом стохастического градиентного спуска.

В предыдущей главе вы уже познакомились с некоторыми компонентами TensorFlow и Keras: с классом `Variable`, операцией `matmul` и объектом `GradientTape` из TensorFlow; вы создали экземпляры слоев `Dense` в Keras, упаковали их в модель `Sequential` и обучили эту модель с помощью метода `fit()`.

Теперь более подробно рассмотрим реализацию всех этих идей на практике с помощью TensorFlow и Keras.

### 3.5.1. Тензоры-константы и тензоры-переменные

В любых операциях, выполняемых с помощью TensorFlow, участвуют тензоры. Тензоры создаются с некоторыми начальными значениями. Например, можно создать тензор с единицами во всех элементах, с нулями (листинг 3.1) или со случайными значениями (листинг 3.2).

#### Листинг 3.1. Тензоры с единицами и с нулями во всех элементах

```
>>> import tensorflow as tf
>>> x = tf.ones(shape=(2, 1))           ← Эквивалентно вызову
>>> print(x)
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)    np.ones(shape=(2, 1))

>>> x = tf.zeros(shape=(2, 1))          ← Эквивалентно вызову
>>> print(x)
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)    np.zeros(shape=(2, 1))
```

**Листинг 3.2.** Тензоры со случайными значениями в элементах

```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.) ←
>>> print(x)
tf.Tensor(
[[-0.14208166]
 [-0.95319825]
 [ 1.1096532 ]], shape=(3, 1), dtype=float32)
Для создания тензора со случайными значениями используется нормальное
распределение со средним отклонением 0 и стандартным отклонением 1.
Эквивалентно вызову np.random.normal(size=(3, 1), loc=0., scale=1.)
>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.) ←
>>> print(x)
tf.Tensor(
[[0.33779848]
 [0.06692922]
 [0.7749394 ]], shape=(3, 1), dtype=float32)
Для создания тензора со случайными значениями используется
равномерное распределение между 0 и 1. Эквивалентно
вызову np.random.uniform(size=(3, 1), low=0., high=1.)
```

Существенная разница между массивами NumPy и тензорами TensorFlow заключается в том, что тензоры TensorFlow не могут изменяться, они подобны константам. Например, в NumPy можно выполнить следующие операции.

**Листинг 3.3.** Массивы NumPy можно изменять

```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

Если попробовать сделать то же самое с тензором TensorFlow, библиотека сообщит об ошибке: `EagerTensor object does not support item assignment` (Объект EagerTensor не поддерживает присваивание значений элементам).

**Листинг 3.4.** Тензоры TensorFlow не могут изменяться

```
x = tf.ones(shape=(2, 2)) ← | Эта операция потерпит неудачу, потому что
x[0, 0] = 0.               | тензоры не могут изменяться
```

Как вы знаете, чтобы обучить модель, нужно в цикле обновлять ее состояние, представленное набором тензоров. Но если тензоры нельзя изменять, как же тогда происходит обучение? В таком случае используются *переменные*. Для управления изменяемым состоянием в TensorFlow применяется класс `tf.Variable`. Вы уже видели его в реализации цикла обучения в конце главы 2.

Чтобы создать такую переменную, нужно указать какое-то начальное значение, например случайный тензор.

**Листинг 3.5.** Создание переменной TensorFlow

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
>>> print(v)
array([[-0.75133973],
       [-0.4872893 ],
       [ 1.6626885 ]], dtype=float32)>
```

Состояние переменной можно менять с помощью ее метода `assign`, как показано ниже.

**Листинг 3.6.** Присваивание нового значения переменной TensorFlow

```
>>> v.assign(tf.ones((3, 1)))
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

Этот метод применим также к подмножеству элементов.

**Листинг 3.7.** Присваивание новых значений подмножеству элементов переменной TensorFlow

```
>>> v[0, 0].assign(3.)
array([[3.],
       [1.],
       [1.]], dtype=float32)>
```

Аналогично для выполнения операций `+=` и `-=` предлагаются методы `assign_add()` и `assign_sub()`.

**Листинг 3.8.** Пример использования `assign_add()`

```
>>> v.assign_add(tf.ones((3, 1)))
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

## 3.5.2. Операции с тензорами: математические действия в TensorFlow

Так же как NumPy, TensorFlow предлагает большую коллекцию тензорных операций для выражения математических формул. Вот несколько примеров.

**Листинг 3.9.** Некоторые простые математические операции

```
a = tf.ones((2, 2))          Возведение в квадрат
b = tf.square(a)             ←
c = tf.sqrt(a)               Квадратный корень
d = b + c                   Сложение двух тензоров (поэлементное)
e = tf.matmul(a, b)          ←
e *= d                      Произведение двух тензоров
                            (как обсуждалось в главе 2)
                            Умножение двух тензоров
                            (поэлементное)
```

Важно отметить, что каждая из предыдущих операций выполняется немедленно: в любой момент вы можете вывести текущий результат, как в NumPy. Мы называем это *жадным выполнением* (*eager execution*).

### 3.5.3. Второй взгляд на GradientTape

Пока что TensorFlow кажется очень похожим на NumPy. Но вот кое-что, чего NumPy не умеет делать: вычисление градиента любого дифференцируемого выражения по отношению к любому из его входов. Просто откройте контекст `GradientTape`, примените некоторые вычисления к одному или нескольким входным тензорам — и получите градиент результата относительно входов.

#### Листинг 3.10. Пример использования GradientTape

```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

Этот прием чаще всего используется для получения градиентов потерь модели относительно ее весов: `gradient = tape.gradient(loss, weights)`. Вы уже видели, как это делается, в главе 2.

До сих пор мы рассматривали только случай, когда входными тензорами в `tape.gradient()` были переменные TensorFlow. Входные данные могут быть представлены любым тензором, но по умолчанию отслеживаются только *обучаемые переменные*. Чтобы задействовать тензор-константу, придется вручную отметить его как отслеживаемый вызовом `tape.watch()`.

#### Листинг 3.11. Пример использования GradientTape с входным тензором-константой

```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
    tape.watch(input_const)
    result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

Почему так? Потому что было бы слишком дорого хранить информацию, необходимую для вычисления градиента чего-либо по отношению к чему-либо. Чтобы не тратить ресурсы впустую, объект `GradientTape` должен знать, за чем наблюдать. Обычно `GradientTape` используется для вычисления градиента потерь относительно списка обучаемых переменных, так что по умолчанию наблюдение ведется именно за обучаемыми переменными.

`GradientTape` — мощный объект, способный даже вычислять *градиенты второго порядка*, то есть градиенты градиентов. Например, градиент положения объекта относительно времени — это скорость объекта, а градиент второго порядка — его ускорение.

Если измерить положение падающего яблока вдоль вертикальной оси с течением времени и обнаружить, что результаты соответствуют формуле `position(time) = 4.9 * time ** 2`, как отсюда получить ускорение? Давайте воспользуемся двумя вложенными контекстами `GradientTape` и выясним это.

#### Листинг 3.12. Использование вложенных контекстов `GradientTape` для вычисления градиента второго порядка

```
time = tf.Variable(0.)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position = 4.9 * time ** 2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

Мы использовали внешний объект `GradientTape` для вычисления градиента из градиента внутреннего объекта `GradientTape`. Естественно, результат получился равным  $4.9 \cdot 2 = 9.8$

### 3.5.4. Полный пример: линейный классификатор на TensorFlow

Вы познакомились с тензорами, переменными и тензорными операциями и узнали, как вычислять градиенты. Оказывается, этого достаточно, чтобы построить любую модель машинного обучения на основе градиентного спуска. А ведь вы только начали читать главу 3!

На собеседовании по машинному обучению вас могут попросить реализовать линейный классификатор с нуля в TensorFlow: очень простая задача, которая помогает отделить кандидатов с минимальным опытом машинного обучения от тех, кто такого опыта не имеет. Давайте вместе выполним это задание и воспользуемся для этого новыми знаниями в TensorFlow.

Для начала создадим искусственный набор данных, включающий два линейно разделимых класса точек на двумерной плоскости. Для этого сгенерируем каждый класс точек, извлекая их координаты из случайного распределения с определенной ковариационной матрицей и определенным средним значением. Ковариационная матрица описывает форму облака точек, а среднее значение — его положение на плоскости (рис. 3.6). Для создания двух облаков точек мы используем одну и ту же ковариационную матрицу, но разные средние значения: как результат, облака точек будут иметь одинаковую форму, но разные местоположения.

**Листинг 3.13.** Создание набора случайных точек двух классов на двумерной плоскости

```
num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
```

Сгенерировать 1000 случайных точек первого класса. cov=[[1, 0.5],[0.5, 1]] соответствует облаку точек овальной формы, вытянутому в направлении от левого нижнего к правому верхнему углу

Сгенерировать точки второго класса с той же ковариационной матрицей, но другим средним значением

Здесь `negative_samples` и `positive_samples` — это массивы с формой (1000, 2). Объединим их в один массив с формой (2000, 2).

**Листинг 3.14.** Объединение точек двух классов в один массив с формой (2000, 2)

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

Теперь сгенерируем соответствующие целевые метки, массив нулей и единиц с формой (2000, 1), где элементы `targets[i, 0]` равны 0, если `input[i]` принадлежит классу 0 (и наоборот).

**Листинг 3.15.** Создание целевых меток (0 или 1)

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

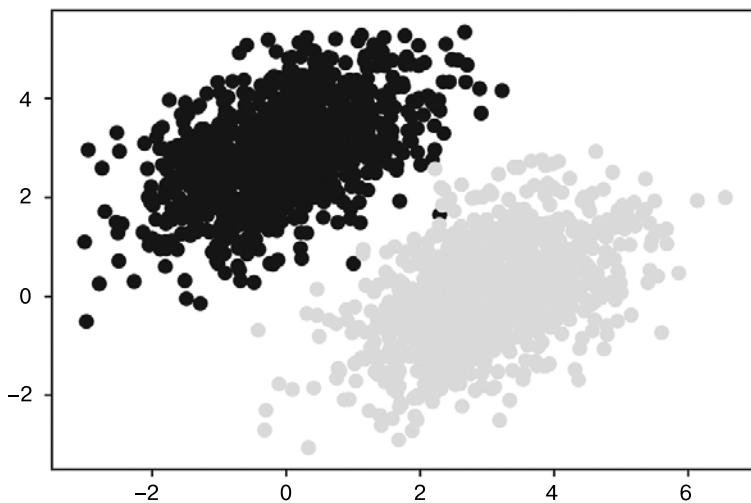
Теперь нарисуем точки с помощью Matplotlib.

**Листинг 3.16.** Вывод классов точек на плоскости (рис. 3.6)

```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```

Теперь создадим линейный классификатор, который научится разделять эти два облака. Линейный классификатор — это аффинное преобразование ( $\text{prediction} = \mathbf{W} \cdot \mathbf{input} + \mathbf{b}$ ), обученное минимизировать квадрат разницы между предсказаниями и целями.

Как вы убедитесь позднее, данный пример на самом деле гораздо проще, чем двухслойная нейронная сеть, которую вы видели в конце главы 2. Но на этот раз у вас достаточно знаний, чтобы понять весь код, каждую его строку.



**Рис. 3.6.** Наши искусственные данные: два класса случайных точек на двумерной плоскости

Давайте создадим переменные  $W$  и  $b$ , инициализированные случайными значениями и нулями соответственно.

#### Листинг 3.17. Создание переменных для линейного классификатора

```
input_dim = 2      ← На вход подаются
output_dim = 1    ← двумерные точки
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

Прогноз на выходе — единственная оценка для каждого образца (близкая к 0, если предполагается, что образец относится к классу 0, или к 1, если предполагается, что образец относится к классу 1)

Вот функция прямого прохода.

#### Листинг 3.18. Функция прямого прохода

```
def model(inputs):
    return tf.matmul(inputs, W) + b
```

Наш линейный классификатор будет работать с двумерными входными данными, поэтому  $W$  на самом деле представляет два скалярных коэффициента,  $w_1$  и  $w_2$ :  $W = [[w_1], [w_2]]$ , а  $b$  — единственный скалярный коэффициент. То есть прогноз для каждой данной входной точки  $[x, y]$  вычисляется так:  $\text{prediction} = [w_1], [w_2] \cdot [x, y] + b = w_1 * x + w_2 * y + b$ .

В следующем листинге показана наша функция потерь.

#### Листинг 3.19. Функция потерь, вычисляющая средний квадрат ошибок

```
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions) ←
    return tf.reduce_mean(per_sample_losses) ←
```

Тензор `per_sample_losses` имеет ту же форму, что и тензоры `targets` и `predictions`, и содержит оценки потерь для каждого образца

Нам нужно усреднить оценки потерь по образцам в одно скалярное значение потерь: именно это делает `reduce_mean`

Далее следует этап обучения, который принимает некоторые обучающие данные и обновляет веса  $W$  и  $b$ , стремясь минимизировать потери на данных.

#### Листинг 3.20. Функция этапа обучения

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(predictions, targets)
        grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b]) ←
        W.assign_sub(grad_loss_wrt_W * learning_rate) ←
        b.assign_sub(grad_loss_wrt_b * learning_rate) ←
    return loss
```

Получение градиента потерь относительно весов

Прямой проход внутри контекста `GradientTape`

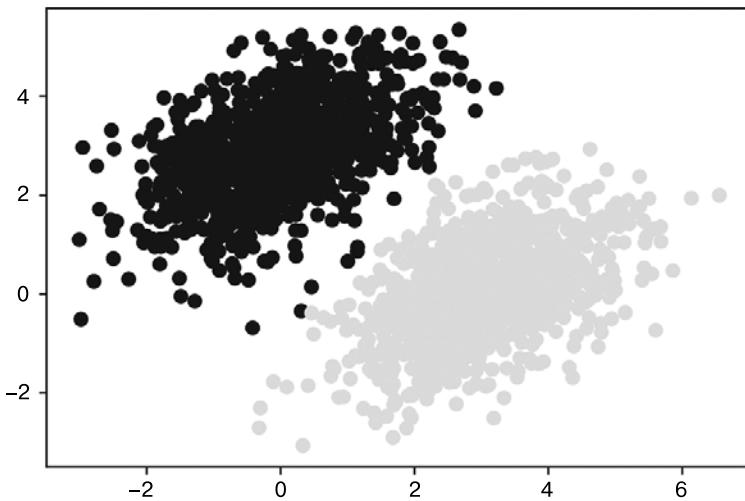
Обновление весов

Для простоты используем *пакетное обучение* вместо *мини-пакетного*: будем запускать каждый шаг обучения (вычисление градиента и обновление весов) сразу для всех данных, не перебирая их небольшими партиями. С одной стороны, это означает, что каждый шаг обучения будет занимать гораздо больше времени, поскольку прямой проход и вычисление градиентов будут производиться для 2000 образцов одновременно. С другой стороны, с каждым новым обновлением градиента потери на обучающих данных будут снижаться намного эффективнее, ведь в расчетах будут участвовать сразу все образцы, а не, скажем, 128 случайно отобранных. В результате потребуется намного меньше шагов обучения и можно взять более высокую скорость обучения, чем при обычном обучении на небольших пакетах (мы используем `learning_rate = 0.1`, как определено в листинге 3.20).

#### Листинг 3.21. Цикл пакетного обучения

```
for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

После 40 циклов обучения величина потерь стабилизировалась на уровне около 0,025. Посмотрим, как получившаяся линейная модель классифицирует точки из обучающего набора данных. Поскольку целевыми значениями у нас служат нули и единицы, всякая входная точка будет классифицироваться как 0, если прогнозируемое значение для нее ниже 0,5, и как 1, если больше 0,5 (рис. 3.7).

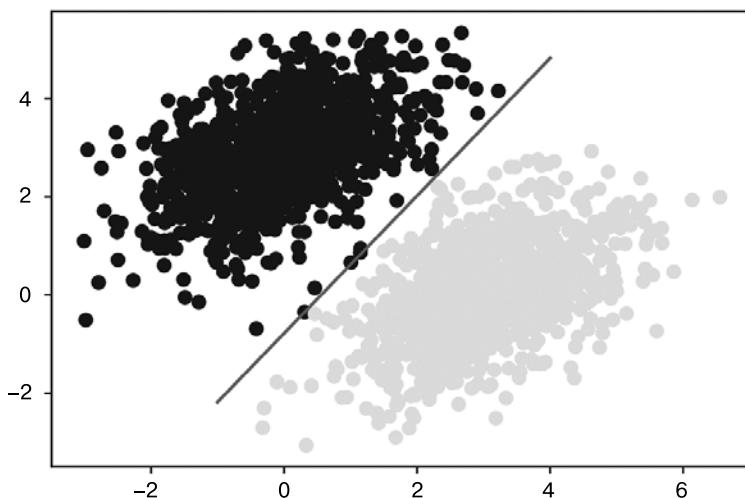


**Рис. 3.7.** Прогноз нашей модели очень близок к исходной картине

Напомню, что значение прогноза для данной точки  $[x, y]$  вычисляется как  $\text{prediction} == [[w_1], [w_2]] \cdot [x, y] + b == w_1 * x + w_2 * y + b$ . То есть считается, что точка принадлежит классу 0, если выполняется условие  $w_1 * x + w_2 * y + b \leq 0.5$ , и классу 1, если выполняется условие  $w_1 * x + w_2 * y + b > 0.5$ . Обратите внимание, что на самом деле перед нами уравнение прямой на двумерной плоскости:  $w_1 * x + w_2 * y + b = 0.5$ . Над линией находятся точки, принадлежащие классу 1, а под линией — принадлежащие классу 0. Если вы привыкли видеть линейные уравнения в формате  $y = a * x + b$ , то уравнение нашей линии можно выразить так:  $y = -w_1 / w_2 * x + (0.5 - b) / w_2$ .

Построим эту линию (рис. 3.8):

```
Сгенерировать 100 чисел, равномерно
распределенных в интервале от -1 до 4, которые
будут использоваться для рисования прямой
x = np.linspace(-1, 4, 100)
Это уравнение
нашей прямой
y = - W[0] / W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
Нарисовать линию ("r" означает
красный (red) цвет)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
Тут же нарисовать прогноз
нашей модели
```



**Рис. 3.8.** Наша модель изображена как прямая линия

В этом и заключается суть линейного классификатора: поиск параметров линии (или, в многомерных пространствах, гиперплоскости), аккуратно разделяющей два класса данных.

## 3.6. АНАТОМИЯ НЕЙРОННОЙ СЕТИ: ЗНАКОМСТВО С ОСНОВАМИ KERAS

Теперь, зная основы библиотеки TensorFlow, вы сможете реализовать с нуля простую модель линейного классификатора, подобную той, что была показана в предыдущем разделе, или несложной нейронной сети, представленной в конце главы 2. Это солидный фундамент, на котором уже можно что-то построить. А сейчас пришло время встать на более продуктивный и надежный путь к глубокому обучению: начать использовать библиотеку Keras.

### 3.6.1. 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: есть *веса* слоя — один или несколько тензоров, обучаемых с применением алгоритма стохастического градиентного спуска, которые вместе хранят *знания*, накапливаемые сетью.

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`, или одномерными сверточными слоями (`Conv1D`). Изображения хранятся в четырехмерных тензорах и обычно обрабатываются двумерными сверточными слоями (`Conv2D`).

Слои можно считать кубиками лего глубокого обучения. Библиотеки, подобные Keras, делают это сравнение еще более явным: создание моделей глубокого обучения в Keras осуществляется путем объединения совместимых слоев в конвейеры обработки данных.

### Базовый класс `Layer` в Keras

Простой прикладной интерфейс (API) должен иметь единую абстракцию, лежащую в основе всего. В Keras такой абстракцией служит класс слоев `Layer`. Все в Keras является либо слоем `Layer`, либо чем-то еще, что тесно взаимодействует со слоем `Layer`.

Слой — это объект, инкапсулирующий некоторое состояние (веса) и некоторые вычисления (прямой проход). Веса обычно определяются с помощью метода `build()` (но также могут инициализироваться в конструкторе `__init__()`), а вычисления определяются в методе `call()`.

В предыдущей главе мы реализовали класс `NaiveDense`, содержащий два веса, `W` и `b`, и применили вычисления `output = activation(dot(input, W) + b)`. Вот как тот же слой выглядел бы в Keras.

#### Листинг 3.22. Слой `Dense`, реализованный как подкласс класса `Layer`

```
from tensorflow import keras
class SimpleDense(keras.layers.Layer):
    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation
    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                               initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,), initializer="zeros")
    def call(self, inputs):
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

Все классы слоев в Keras наследуют базовый класс `Layer`

`add_weight()` — это вспомогательный метод для создания весов. Также имеется возможность создать отдельные переменные и связать их с атрибутами слоя, например:  
`self.W = tf.Variable(tf.random.uniform(w_shape))`

Веса создаются в методе `build()`

Вычисления, выполняемые во время прямого прохода, определяются в методе `call()`

Мы еще вернемся к методам `build()` и `call()` в следующем разделе и рассмотрим их подробнее, поэтому не волнуйтесь, если вы чего-то не поняли!

Создав такой слой, его можно использовать как функцию, принимающую на входе тензор TensorFlow:

```
>>> my_dense = SimpleDense(units=32, activation=tf.nn.relu)
>>> input_tensor = tf.ones(shape=(2, 784))
>>> output_tensor = my_dense(input_tensor) ←
>>> print(output_tensor.shape)
(2, 32)
```

Вам, наверное, интересно узнать, зачем нужно было реализовать методы `call()` и `build()`, если в итоге мы использовали наш слой, просто вызвав его как функцию, то есть с помощью его метода `__call__()`? Причина проста: нам нужно, чтобы состояние создавалось динамически, на лету. Давайте посмотрим, как это работает.

### Автоматическое определение формы: построение слоев на лету

Подобно кубикам лего, состыковать можно только совместимые слои. Понятие *совместимости слоев* в нашем случае отражает лишь тот факт, что каждый слой принимает и возвращает тензоры определенной формы. Взгляните на следующий пример:

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation="relu") ←
```

Слой возвращает тензор, первое измерение которого равно 32. Данный слой можно связать со слоем ниже, только если тот принимает 32-мерные векторы.

В большинстве случаев библиотека Keras избавляет от необходимости беспокоиться о совместимости, поскольку слои, добавляемые в модели, автоматически конструируются так, чтобы соответствовать форме входного слоя. Представьте, что вы написали следующий код:

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32)
])
```

Слои не получают никакой информации о форме входных данных — они автоматически определяют ее по форме первого измерения своих входных данных.

В упрощенной версии слоя `Dense`, реализованной нами в главе 2 (и названной `NaiveDense`), требовалось явно передать размер входных данных конструктору, чтобы получить возможность создать веса. Это не очень удобно, поскольку тогда при конструировании моделей мы вынуждены будем явно указывать в каждом новом слое форму выходных данных предыдущего слоя:

```
model = NaiveSequential([
    NaiveDense(input_size=784, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=64, activation="relu"),
    NaiveDense(input_size=64, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=10, activation="softmax")
])
```

Было бы еще хуже, если бы при выборе формы своих выходных данных слой руководствовался сложными правилами. Допустим, наш слой возвращает выходные данные с формой (`batch, input_size * 2 if input_size % 2 == 0 else input_size * 3`).

Если бы мы повторно реализовали слой `NaiveDense` как слой Keras, поддерживающий автоматическое определение формы входных данных, то он выглядел бы как предыдущий слой `SimpleDense` (см. листинг 3.22) с его методами `build()` и `call()`.

В `SimpleDense` мы не создаем веса в конструкторе, как это делали в примере `NaiveDense`; теперь они создаются в специальном методе конструирования состояния `build()`, который принимает в аргументе форму первого измерения входных данных. Метод `build()` вызывается автоматически при первом вызове слоя (через метод `__call__()`). Именно поэтому мы определили вычисления в отдельном методе `call()`, а не в методе `__call__()` непосредственно. В общих чертах метод `__call__()` базового слоя выглядит примерно так:

```
def __call__(self, inputs):
    if not self.built:
        self.build(inputs.shape)
        self.built = True
    return self.call(inputs)
```

Благодаря автоматическому определению формы наш предыдущий пример становится простым и понятным:

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

Обратите внимание: автоматическое определение формы не единственное, что может метод `__call__()` класса `Layer`. Он также решает множество других задач, в частности делает выбор между *жадным* (немедленным) и *графовым* (с этим понятием вы познакомитесь в главе 7) выполнением, а также накладывает маску на входные данные (об этом рассказывается в главе 11). Пока просто запомните: приступая к реализации собственных слоев, описывайте прямой проход в методе `call()`.

### 3.6.2. От слоев к моделям

Модель глубокого обучения является графом слоев. В Keras модели представляют собой экземпляры класса `Model`. К настоящему моменту вы видели только последовательные модели `Sequential` (подкласс класса `Model`) — простой 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).

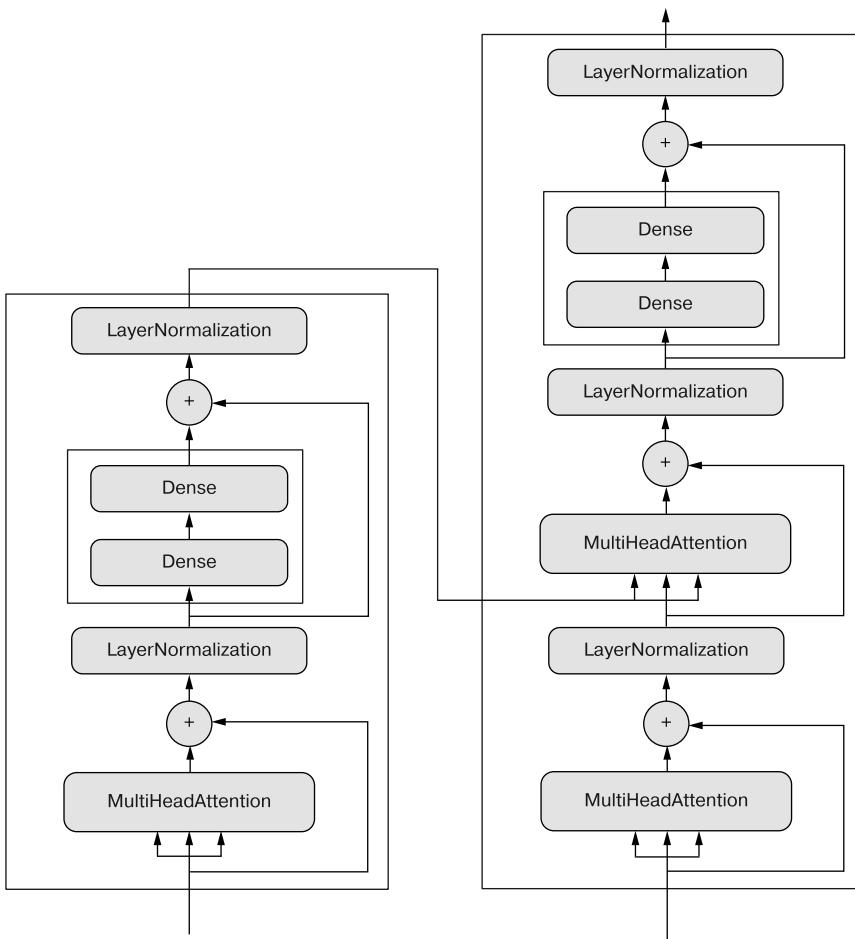
Сети могут иметь весьма сложную топологию. Например, на рис. 3.9 показана топология графа слоев `Transformer` — универсальной архитектуры для обработки текстовых данных.

Вообще, в Keras существует два способа создания таких моделей: можно напрямую создать подкласс класса `Model` или использовать функциональный API, позволяющий делать больше с меньшим количеством кода. Мы рассмотрим оба подхода в главе 7.

Топология сети определяет *пространство гипотез*. Вспомните, как в главе 1 мы установили, 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.

Для обучения на данных необходимы предположения — они определяют, чему можно обучиться. Поэтому структура пространства гипотез — архитектура модели — чрезвычайно важна. Она кодирует предположения о решаемой задаче, предварительные знания, с которых начинается модель. Например, при работе

над задачей классификации двух классов выбор модели, состоящей из одного плотного слоя без активации (чистое аффинное преобразование), предполагает, что два класса линейно разделимы.



**Рис. 3.9.** Архитектура Transformer (будет рассмотрена подробнее в главе 11). Здесь много интересного. В следующих нескольких главах вы приблизитесь к ее пониманию

Выбор правильной архитектуры сети — больше искусства, чем науки; и хотя есть несколько проверенных методов и принципов, на которые можно положиться, только практика может помочь вам стать опытным архитектором нейронных сетей. В следующих главах вы познакомитесь с отдельными принципами

конструирования нейронных сетей и получите базовое представление о том, что подходит или не подходит для решения конкретных задач. Вы поймете, какие типы архитектур моделей пригодны для тех или иных задач, как сконструировать эти модели на практике, как выбрать правильную конфигурацию обучения и как настроить модель, чтобы получить желаемые результаты.

### 3.6.3. Этап «компиляции»: настройка процесса обучения

После того как вы определились с архитектурой сети, нужно выбрать еще три параметра:

- *функцию потерь* (*целевую функцию*) — количественную оценку, которая будет минимизироваться в процессе обучения. Представляет собой меру успеха в решении стоящей задачи;
- *оптимизатор* — определяет, как будет изменяться сеть под воздействием функции потерь. Реализует конкретный вариант стохастического градиентного спуска (Stochastic Gradient Descent, SGD);
- *метрики* — показатели успеха (такие как точность классификации), за которыми будет вестись наблюдение во время обучения и проверки. Обучение, в отличие от потерь, не оптимизируется по данным показателям напрямую. Поэтому от метрик не требуется, чтобы они были дифференцированными.

После выбора функции потерь, оптимизатора и метрик можно использовать встроенные методы `compile()` и `fit()`, чтобы начать обучение модели. При желании можно также реализовать собственные циклы обучения — мы расскажем, как это сделать, в главе 7. Придется приложить очень много усилий! Пока же давайте взглянем на `compile()` и `fit()`.

Метод `compile()` настраивает процесс обучения — вы уже познакомились с ним в самом первом примере нейронной сети в главе 2. Он принимает аргументы с оптимизатором, функцией потерь и метриками (в виде списка):

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer="rmsprop",
              loss="mean_squared_error",
              metrics=["accuracy"])

Определение линейного
классификатора
Определение оптимизатора
по имени: RMSprop (регистр
имеет значение)
Определение
функции по имени:
среднеквадратичная ошибка
Определение списка метрик: в данном
случае только accuracy (точность)
```

В этом примере методу `compile()` оптимизатор, функция потерь и список метрик передаются в виде строковых имен (например, "rmsprop").

В действительности данные строки являются ярлыками, преобразующими-ся в объекты Python. Например, "rmsprop" превращается в `keras.optimizers.RMSprop()`. Важно помнить, что вместо строк можно также передать экземпляры объектов, например:

```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

Это удобно тем, кто хочет использовать собственные функции потерь или метрики или желает выполнить дополнительную настройку применяемых объектов, например передать аргумент `learning_rate` оптимизатору:

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
              loss=my_custom_loss,
              metrics=[my_custom_metric_1, my_custom_metric_2])
```

В главе 7 вы узнаете, как создавать свои функции потерь и метрики. В общем случае нет необходимости прописывать функции потерь, метрики или оптимизаторы с нуля, поскольку Keras предлагает широкий спектр встроенных опций, среди которых наверняка найдется то, что вам нужно:

- оптимизаторы:
  - SGD (с импульсом или без);
  - RMSprop;
  - Adam;
  - Adagrad и др.;
- функции потерь:
  - CategoricalCrossentropy;
  - SparseCategoricalCrossentropy;
  - BinaryCrossentropy;
  - MeanSquaredError;
  - KLDivergence;
  - CosineSimilarity и др.;
- метрики:
  - CategoricalAccuracy;
  - SparseCategoricalAccuracy;
  - BinaryAccuracy;

- AUC;
- Precision;
- Recall и др.

Далее в книге вы увидите многие из этих вариантов в действии.

### 3.6.4. Выбор функции потерь

Выбор правильной функции потерь для решения конкретной задачи играет очень важную роль: ваша модель будет использовать любую возможность минимизировать потери, поэтому если функция потерь не отвечает полностью критериям успешного решения задачи, то модель в конечном счете может выдать совсем не тот результат, 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, кроме нескольких человек, и сосредоточиться на их потребностях, поскольку среднее благосостояние не зависит от количества оставшихся. Но мы же совсем не это имели в виду! Нейронные сети, которые вы строите, в минимизации функции потерь будут столь же беспощадны, поэтому мудро выбирайте цель, иначе вам придется столкнуться с неожиданными побочными эффектами.

К счастью, для типовых задач (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, в несколько категорий — многозначной перекрестной энтропии и т. д. Свои функции потерь вам придется разрабатывать, только сталкиваясь с действительно новыми исследовательскими задачами. В следующих нескольких главах мы подробно объясним, какие функции потерь стоит выбирать для распространенных задач.

### 3.6.5. Метод fit()

За вызовом `compile()` следует вызов `fit()`. Метод `fit()` реализует собственно цикл обучения. Вот его основные аргументы:

- *данные* для обучения (исходные данные и целевые значения). Обычно передаются в виде массивов NumPy или объекта `Dataset` из библиотеки TensorFlow (больше о возможностях `Dataset` вы узнаете в следующих главах);
- количество эпох обучения: сколько раз должен повторяться цикл обучения на переданных данных;

- размер пакета для использования в каждой эпохе обучения методом градиентного спуска: количество обучающих образцов, учитываемых при вычислении градиентов в одном шаге обновления весов.

### Листинг 3.23. Вызов метода fit() с данными в формате NumPy

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
    
```

Исходные образцы  
в виде массива NumPy

Цели обучения в виде  
массива NumPy

Цикл обучения выполнит  
пять итераций по данным

В итерациях цикла обучения исходные данные  
будут обрабатываться пакетами по 128 образцов

Вызов `fit()` возвращает объект `History` с полем `history` — словарем, ключами которого служат имена метрик или строки (такие как `"loss"`), а значениями — списки значений соответствующих метрик, полученных в разные эпохи.

```
>>> history.history
{"binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],
 "loss": [0.6573270302042366,
          0.07434618508815766,
          0.07687718723714351,
          0.07412414988875389,
          0.07617757616937161]}
```

## 3.6.6. Оценка потерь и метрик на проверочных данных

Цель машинного обучения не в том, чтобы создать модели, которые дают точные прогнозы на обучающих данных (что довольно просто — достаточно лишь следовать за градиентом), а в том, чтобы создать модель, хорошо справляющуюся со своей задачей в целом — и особенно на данных, которые она раньше не видела. Хорошие результаты на обучающих данных не гарантируют такой же исход на данных, которые модель не видела прежде! Например, она может просто *запомнить* связь между обучающими данными и ожидаемыми результатами — и в новых условиях для задачи прогнозирования станет совершенно бесполезной. Мы рассмотрим этот аспект более подробно в главе 5.

Для оценки качества модели — того, как она справляется со своей задачей на новых данных, — обычно принято выделять некоторую часть исходных данных в отдельную *проверочную выборку*: данные из этой выборки не участвуют в обучении модели, но используются для вычисления величины потерь и метрик. Проверочную выборку можно передать методу `fit()` в аргументе `validation_data`.

По аналогии с обучающими данными проверочные данные могут передаваться в форме массива NumPy или объекта `Dataset` из библиотеки TensorFlow.

#### Листинг 3.24. Использование аргумента validation\_data

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets) ←
)

```

Чтобы избежать попадания в проверочную выборку только экземпляров одного класса, исходные и целевые данные перемешиваются методом случайной перестановки индексов

Зарезервировать 30 % исходных и целевых данных для проверки (эти образцы будут исключены из процесса обучения и используются только для вычисления величины потерь и метрик)

Проверочные данные, использующиеся только для оценки величины потерь и метрик на этапе проверки

Величина потерь, полученная при оценке на проверочных данных, называется *потерей на проверочных данных*, чтобы отличать ее от *потери на обучающих данных*. Помните: важно строго отделять одни данные от других. Задача проверки состоит в том, чтобы оценить, насколько хорошие результаты показывает обученная модель на новых данных. Если модель видела проверочные данные во время обучения, то потери на проверочных данных и метрики будут оцениваться некорректно.

Потери на проверочных данных и метрики можно вычислить после завершения обучения вызовом метода `evaluate()`:

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

Метод `evaluate()` выполнит итерации по пакетам (размером `batch_size`) с переданными данными и вернет список скаляров, первый из которых — величина потерь на проверочных данных, а последующие — метрики. Если модель не имеет метрик, то возвращено будет только одно значение — величина потерь на проверочных данных (а не список).

### 3.6.7. Вывод: использование модели после обучения

После обучения модель можно использовать для вычисления прогнозов на основе новых данных. Этот этап называется *выводом*. Простейший способ получить прогноз — вызвать метод `__call__()` модели:

```
predictions = model(new_inputs) ←
    Принимает массив NumPy или тензор
    TensorFlow и возвращает тензор TensorFlow
```

Однако это подразумевает обработку сразу всех входных данных в `new_inputs`, что может оказаться невыполнимым, если, например, объем данных для прогнозирования слишком большой и для его обработки требуется больше памяти, чем имеется у вашего графического процессора.

Лучший способ получить вывод — использовать метод `predict()`. Он выполнит обход данных, разбив их на небольшие пакеты, и вернет массив NumPy с прогнозами. В отличие от `__call__()`, он также может обрабатывать объекты `Dataset`.

```
predictions = model.predict(new_inputs, batch_size=128) ←
    Принимает массив NumPy или объект
    Dataset и возвращает массив NumPy
```

Например, если к некоторым из проверочных данных применить метод `predict()` обученной выше модели линейной классификации, то он вернет скалярные оценки, соответствующие прогнозу модели для каждого входного образца:

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10])
[[0.3590725]
 [0.82706255]
 [0.74428225]
 [0.682058]
 [0.7312616]
 [0.6059811]
 [0.78046083]
 [0.025846]
 [0.16594526]
 [0.72068727]]
```

На данный момент это все, что нужно знать о моделях Keras. Теперь вы готовы перейти к решению реальных задач машинного обучения с помощью Keras, чем мы и займемся в следующей главе.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- TensorFlow — мощный фреймворк для числовых вычислений, который может работать на CPU, GPU или TPU. Он способен автоматически вычислять градиент любого дифференцируемого выражения, распределять работу среди множества устройств и экспортить программы в различные внешние среды выполнения — даже в JavaScript.
- Keras — стандартная библиотека, используемая для глубокого обучения с помощью TensorFlow. Именно ее мы будем применять в этой книге.
- К ключевым компонентам TensorFlow относятся тензоры, переменные, тензорные операции и объект `GradientTape`.
- Центральный класс в библиотеке Keras — `Layer` (представляющий слой). Слой инкапсулирует веса и вычисления. Из слоев конструируются модели.
- Прежде чем начать обучение модели, нужно выбрать *оптимизатор, функцию потерь* и метрики и передать их методу `model.compile()`.
- Для обучения модели можно использовать метод `fit()`, который производит обучение методом градиентного спуска на мини-пакетах. Он также будет полезен для оценки величины потерь и метрик на *проверочных данных* — выборке из исходных данных, которая не участвует в процессе обучения модели.
- После обучения модель можно использовать для вычисления прогнозов, вызывая ее метод `model.predict()` с новыми входными данными.

# *Начало работы с нейронными сетями: классификация и регрессия*

## **В этой главе**

- ✓ Первые примеры решения реальных задач машинного обучения.
- ✓ Решение задач классификации векторных данных.
- ✓ Решение задач непрерывной регрессии для векторных данных.

Цель данной главы — помочь вам начать использовать нейронные сети для решения практических задач. Здесь вы закрепите знания, приобретенные в главах 2 и 3, и примените их в трех новых задачах, охватывающих наиболее типичные случаи использования нейронных сетей:

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

Эти примеры познакомят вас со всеми этапами процесса машинного обучения: с предварительной обработкой данных, основными принципами выбора архитектуры модели и оценкой модели.

## ГЛОССАРИЙ КЛАССИФИКАЦИИ И РЕГРЕССИИ

В классификации и регрессии используется множество специальных терминов. Некоторые из них уже встречались вам в предыдущих примерах; еще больше их появится в следующих главах. Все они имеют точные, специфичные для машинного обучения определения, 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 *величина 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», то «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, 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).
- *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.

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 5.

## 4.1. 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.

### 4.1.1. Набор данных IMDB

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.

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. Это позволит нам сосредоточиться на конструировании моделей, их обучении и оценке. В главе 11 вы узнаете, как использовать необработанные текстовые данные с нуля.

Код в листинге 4.1 загружает набор данных (при первом запуске на ваш компьютер будет загружено примерно 80 Мбайт данных).

#### Листинг 4.1. Загрузка набора данных IMDB

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

Аргумент `num_words=10000` означает, что будет сохранено только 10 000 слов, наиболее часто встречающихся в обучающем наборе отзывов. Редкие слова будут отброшены. Это позволит вам работать с вектором управляемого размера. Если не установить данный предел, то модели придется столкнуться с 88 585 уникальными словами — это слишком много. Многие из них встречаются только в одном образце и поэтому не несут полезной информации для классификации.

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, в листинге 4.2 показан пример декодирования одного из отзывов в последовательность слов на английском языке.

#### Листинг 4.2. Декодирование отзыва обратно в текст

```
word_index = imdb.get_word_index() ← word_index — это словарь, отображающий
reverse_word_index = dict(             слова в целочисленные индексы
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

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

Получить обратное представление словаря, отображающее индексы в слова

### 4.1.2. Подготовка данных

Нельзя передать списки целых чисел непосредственно в нейронную сеть. Все они имеют разную длину, тогда как нейронная сеть ожидает получить для обработки согласованные пакеты данных. 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, максимальная\_длина) и затем передать их в первый слой модели, 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 [8, 5] в 10 000-мерный вектор, все элементы которого содержат нули, кроме элементов с индексами 8 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 для лучшего понимания предмета создадим вручную.

**Листинг 4.3.** Кодирование последовательностей целых чисел в бинарную матрицу

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension)) ← Создание матрицы с формой
    for i, sequence in enumerate(sequences):           (len(sequences), dimension),
                                                       заполненной нулями
        for j in sequence:                            ← Запись единицы в элемент
            results[i, j] = 1.                         с данным индексом
    return results
x_train = vectorize_sequences(train_data)           ← Векторизованные
x_test = vectorize_sequences(test_data)             ← Векторизованные
                                                    ← Векторизованные
                                                    ← Векторизованные
                                                    ← Векторизованные
                                                    ← Векторизованные
```

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.

### 4.1.3. Конструирование модели

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

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

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

В главе 5 вы познакомитесь с формальными принципами, помогающими сделать правильный выбор. А пока вам остается только довериться мне:

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

На рис. 4.1 показано, как выглядит модель. Реализация этой модели с использованием Keras отражена в листинге 4.4 — она напоминает пример MNIST, который мы видели раньше.

#### Листинг 4.4. Определение модели

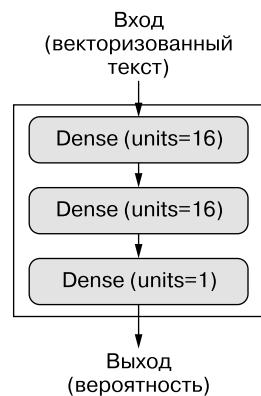
```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

Первым аргументом каждому слою `Dense` передается количество нейронов в этом слое: размерность пространства представления слоя. Как рассказывалось в главах 2 и 3, каждый такой слой `Dense` с функцией активации `relu` реализует следующую цепочку тензорных операций:

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

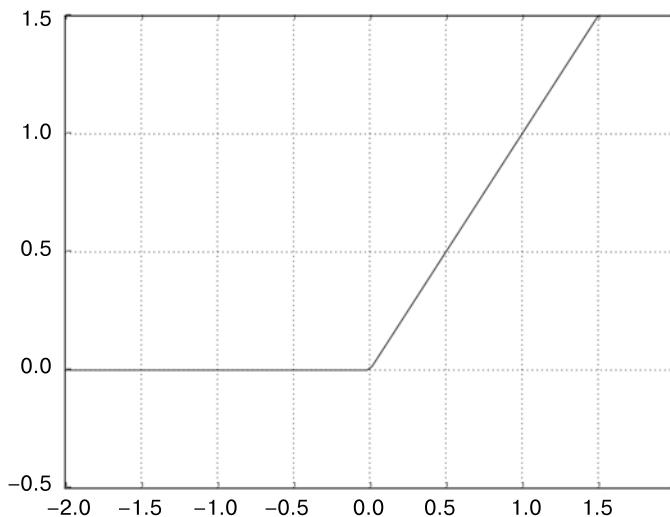
Наличие 16 нейронов означает, что весовая матрица `W` будет иметь форму (`input_dimension, 16`): скалярное произведение на `W` спроектирует входные данные в 16-мерное пространство представлений (затем будет произведено сложение с вектором смещений `b` и выполнена операция `relu`). Размерность



**Рис. 4.1.** Трехслойная модель

пространства представлений можно интерпретировать как «степень свободы модели при изучении внутренних представлений». Большее количество скрытых нейронов (большая размерность пространства представлений) позволяет модели обучаться на более сложных представлениях, но при этом увеличивается вычислительная стоимость модели, что может привести к выявлению нежелательных шаблонов (которые могут повысить качество классификации обучающих данных, но не контрольных).

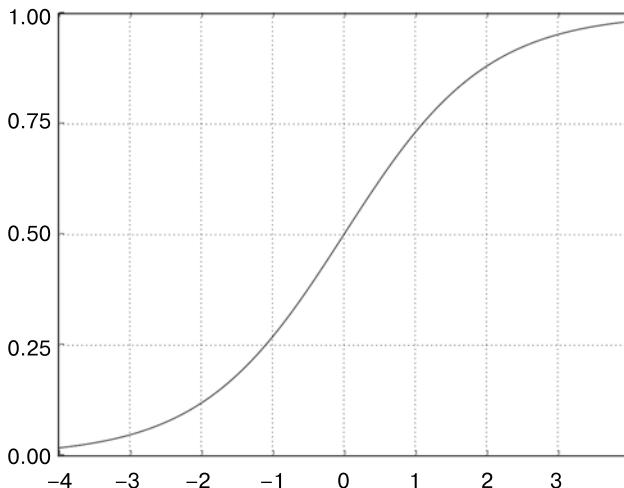
Промежуточному слою понадобится операция `relu` в качестве функции активации, а последний слой будет использовать сигмоидную функцию активации и выводить вероятность (оценку вероятности, между 0 и 1, того, что образец относится к классу 1, то есть насколько он близок к положительному отзыву). Функция `relu` (rectified linear unit – блок линейной ректификации) используется для преобразования отрицательных значений в ноль (рис. 4.2), а сигмоидная функция рассредоточивает произвольные значения по интервалу  $[0, 1]$  (рис. 4.3), возвращая значения, которые можно интерпретировать как вероятность.



**Рис. 4.2.** Функция блока линейной ректификации

Наконец, нужно выбрать функцию потерь и оптимизатор. Так как перед нами стоит задача бинарной классификации и результатом работы модели является вероятность (наша модель заканчивается слоем с единственным нейроном и сигмоидной функцией активации), предпочтительнее использовать функцию потерь `binary_crossentropy`. Это не единственный приемлемый выбор: можно также задействовать, например, `mean_squared_error`. Однако перекрестная энтропия обычно предпочтительнее, 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.



**Рис. 4.3.** Сигмоидная функция

#### 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 ограничено, 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 немногим странными именами: `prelu`, `elu` и т. д.

Что касается выбора оптимизатора, то в этой модели мы будем использовать `rmsprop` — хороший вариант по умолчанию для большинства задач.

Теперь настроим модель, передав ей оптимизатор `rmsprop` и функцию потерь `binary_crossentropy`. Обратите внимание, что мы также задали мониторинг точности во время обучения.

#### **Листинг 4.5.** Компиляция модели

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

### **4.1.4. Проверка решения**

Как отмечалось в главе 3, качество модели глубокого обучения никогда не должно оцениваться на обучающих данных — обычно для мониторинга изменения точности модели во время обучения используется проверочная выборка. Создадим такую выборку, включив в нее 10 000 образцов из оригинального набора обучающих данных.

#### **Листинг 4.6.** 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 обучающей выборке) пакетами по 512 образцов. В то же время будем следить за потерями и точностью по 10 000 отложенных образцов. Для этого достаточно передать проверочные данные в аргументе `validation_data`.

#### **Листинг 4.7.** Обучение модели

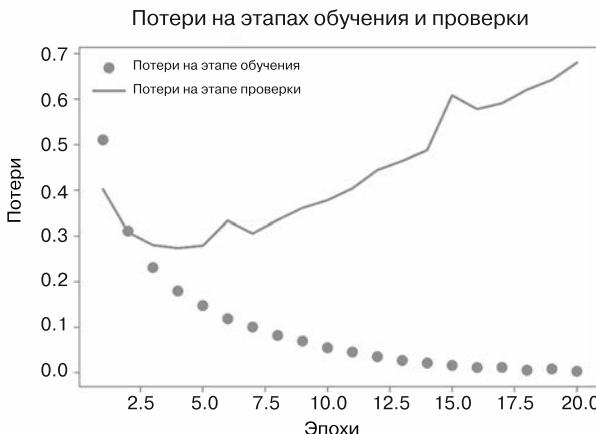
```
history = model.fit(partial_x_train,
                     partial_y_train,
                     epochs=20,
                     batch_size=512,
                     validation_data=(x_val, y_val))
```

При использовании CPU на каждую эпоху будет потрачено менее 2 секунд — а все обучение закончится через 20 секунд. В конце каждой эпохи обучение приостанавливается для вычисления потерь и точности на 10 000 образцах проверочных данных.

Обратите внимание, что вызов `model.fit()` возвращает объект `History`, с которым вы познакомились в главе 3. Этот объект имеет поле `history` — словарь с данными обо всем происходящем в процессе обучения. Заглянем в него:

```
>>> history_dict = history.history
>>> history_dict.keys()
[u"accuracy", u"loss", u"val_accuracy", 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. 4.4) и графиков точности на этапах обучения и проверки (рис. 4.5). Имейте в виду, что ваши результаты могут несколько отличаться: это обусловлено различием в случайных числах, использовавшихся для инициализации сети.



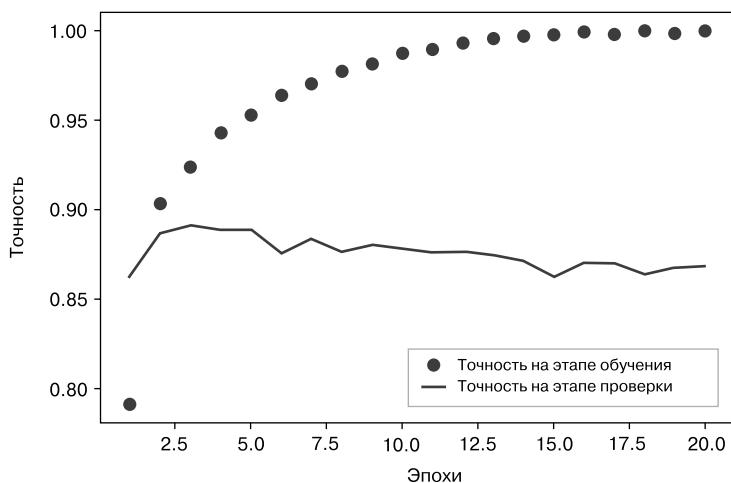
**Рис. 4.4.** Потери на этапах обучения и проверки

#### Листинг 4.8. 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(loss_values) + 1)
plt.plot(epochs, loss_values, "bo", label="Потери на этапе обучения") ←
plt.plot(epochs, val_loss_values, "b", label="Потери на этапе проверки") ←
plt.title("Потери на этапах обучения и проверки")
plt.xlabel("Эпохи")
plt.ylabel("Потери")
plt.legend()
plt.show()
```

bo означает blue dot —  
 «синяя точка»

b означает solid blue line —  
 «сплошная синяя линия»



**Рис. 4.5.** Точность на этапах обучения и проверки

#### **Листинг 4.9.** Формирование графиков точности на этапах обучения и проверки

```
plt.clf() ← Очистить  
рисунок  
acc = history_dict["accuracy"]  
val_acc = history_dict["val_accuracy"]  
plt.plot(epochs, acc, "bo", label="Точность на этапе обучения")  
plt.plot(epochs, val_acc, "b", label="Точность на этапе проверки")  
plt.title("Точность на этапах обучения и проверки")  
plt.xlabel("Эпохи")  
plt.ylabel("Точность")  
plt.legend()  
plt.show()
```

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. Перед вами пример того, о чём мы говорили выше: модель, показывающая хорошие результаты на обучающих данных, не обязательно даст такие же на данных, которые не видела прежде. Выражаясь точнее, в данном случае наблюдается *переобучение*: после четвертой эпохи произошла чрезмерная оптимизация на обучающих данных — в результате получилось представление, характерное для обучающих данных и не обобщающее данные за пределами обучающего набора.

В данном случае для предотвращения переобучения можно прекратить обучение после четвертой эпохи. Вообще, есть целый спектр приемов, ослабляющих эффект переобучения, — их мы рассмотрим в главе 5.

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

#### Листинг 4.10. Обучение новой модели с нуля

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    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] ← | Первое число (0,29) — это потери
                                                | на контрольной выборке; второе число (0,88) —
                                                | точность на контрольной выборке
```

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

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

После обучения модели ее можно использовать для решения практических задач. Например, попробуем оценить 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, модель уверена в одних образцах (0,99 или выше или 0,01 или ниже), но не так уверена в других (0,6; 0,4).

### 4.1.6. 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 слоя, формирующих пространство представлений, перед последним классифицирующим слоем. Попробуйте взять один или три — и посмотрите, как это повлияет на точность на этапах обучения и проверки.
- Попробуйте использовать слои с большим или с меньшим количеством нейронов: 32 нейрона, 64 нейрона и т. д.
- Попробуйте вместо `binary_crossentropy` применить функцию потерь `mse`.
- Попробуйте вместо `relu` использовать функцию активации `tanh` (она была популярна на заре становления нейронных сетей).

### 4.1.7. 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 и другие варианты.
- Стек слоев `Dense` с функцией активации `relu` способен решать широкий круг задач (включая классификацию настроений), и вы, вероятно, чаще всего будете использовать именно эту комбинацию.
- В задаче бинарной классификации (с двумя выходными классами) в конце вашей модели 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 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.
- По мере улучшения результатов на обучающих данных нейронные сети рано или поздно начинают переобучаться, демонстрируя ухудшение на данных, которые они прежде не видели. Поэтому всегда контролируйте качество работы сети на данных не из обучающего набора.

## 4.2. 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 число классов больше двух?

Ниже мы создадим модель 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*. Каждый экземпляр данных должен быть отнесен только к одному классу, поэтому наш пример является *однозначной многоклассовой классификацией*. Если бы каждый экземпляр данных мог принадлежать нескольким классам (в данном случае темам), эта задача была бы примером *многозначной многоклассовой классификации*.

### 4.2.1. 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 представлены более широко, некоторые — менее, но для каждой из них в обучающем наборе имеется не менее десяти примеров.

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

#### Листинг 4.11. Загрузка данных Reuters

```
from tensorflow.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 десятью тысячами наиболее часто встречающихся слов.

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 вам интересно).

#### Листинг 4.12. 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 unknown (неизвестно)

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

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

### 4.2.2. Подготовка данных

Для векторизации данных можно повторно использовать код из предыдущего примера.

#### Листинг 4.13. Кодирование данных

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Векторизованные обучающие данные

Векторизованные контрольные данные

Векторизовать метки можно двумя способами: сохранить их в тензоре целых чисел или использовать *прямое кодирование*. Прямое кодирование (one-hot encoding) широко используется для подготовки категорийных данных и также называется *кодированием категорий* (categorical encoding). В данном случае прямое кодирование меток заключается в конструировании вектора с нулевыми элементами и значением 1 в элементе, индекс которого соответствует индексу метки. Пример приведен в листинге 4.14.

#### Листинг 4.14. Кодирование меток

```
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
y_train = to_one_hot(train_labels)
y_test = to_one_hot(test_labels)
```

Векторизованные обучающие метки

Векторизованные контрольные метки

Следует отметить, что этот способ уже реализован в Keras:

```
from tensorflow.keras.utils import to_categorical  
y_train = to_categorical(train_labels)  
y_test = to_categorical(test_labels)
```

### 4.2.3. Конструирование модели

Задача классификации по темам напоминает предыдущую задачу с отзывами: в обоих случаях мы пытаемся классифицировать короткие фрагменты текста. Но теперь количество выходных классов увеличилось с 2 до 46. Размерность выходного пространства стала 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 для нее бутылочным горлышком. В предыдущем примере мы использовали 16-мерные промежуточные слои, но 16-мерное пространство может оказаться слишком ограниченным для классификации на 46 разных классов: именно 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 4.15. Определение модели

```
model = keras.Sequential([  
    layers.Dense(64, activation="relu"),  
    layers.Dense(64, activation="relu"),  
    layers.Dense(46, activation="softmax")  
])
```

Отметим еще две особенности этой архитектуры.

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

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

#### **Листинг 4.16.** Компиляция модели

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

### 4.2.4. Проверка решения

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

#### **Листинг 4.17.** 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.

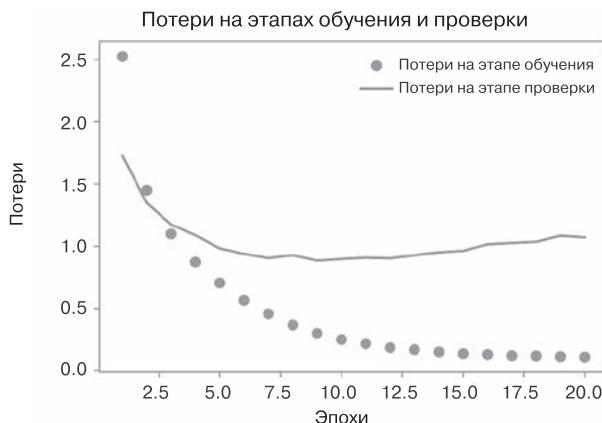
#### **Листинг 4.18.** Обучение модели

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

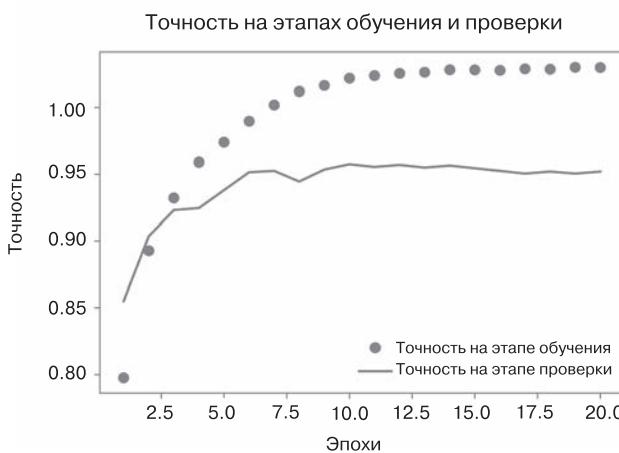
И наконец, выведем графики кривых потерь и точности (рис. 4.6 и 4.7).

#### **Листинг 4.19.** Формирование графиков потерь на этапах обучения и проверки

```
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, "bo", label="Потери на этапе обучения")
plt.plot(epochs, val_loss, "b", label="Потери на этапе проверки")
plt.title("Потери на этапах обучения и проверки")
plt.xlabel("Эпохи")
plt.ylabel("Потери")
plt.legend()
plt.show()
```



**Рис. 4.6.** Потери на этапах обучения и проверки



**Рис. 4.7.** Точность на этапах обучения и проверки

**Листинг 4.20.** Формирование графиков точности на этапах обучения и проверки

```
plt.clf() ← Очистить рисунок
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Точность на этапе обучения")
plt.plot(epochs, val_acc, "b", label="Точность на этапе проверки")
plt.title("Точность на этапах обучения и проверки")
plt.xlabel("Эпохи")
plt.ylabel("Точность")
plt.legend()
plt.show()
```

Переобучение сети наступает после девятой эпохи. Давайте теперь обучим новую модель в течение девяти эпох и затем оценим получившийся результат на контрольных данных.

#### **Листинг 4.21.** Обучение новой модели с нуля

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train,
          y_train,
          epochs=9,
          batch_size=512)
results = model.evaluate(x_test, y_test)
```

Конечные результаты:

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

Данное решение достигло точности ~80 %. В сбалансированной задаче бинарной классификации точность чисто случайного классификатора составила бы 50 %. Но мы имеем 46 классов, и они могут быть представлены неодинаково. Интересно, какую точность дал бы простой случайный классификатор в этом случае? Проверим эмпирически, реализовав такой классификатор на скорую руку:

```
>>> 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)
>>> hits_array.mean()
0.18655387355298308
```

Как видите, случайный классификатор показал точность классификации около 19 % — и в этом свете результаты нашей модели выглядят очень неплохо.

#### **4.2.5. Предсказания на новых данных**

Метод `predict` модели возвращает распределение вероятностей по всем 46 темам для каждого образца. Давайте сгенерируем предсказания для всех контрольных данных.

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

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

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

Сумма коэффициентов этого вектора равна 1:

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

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

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

## 4.2.6. Другой способ обработки меток и потерь

Выше мы упоминали, что метки также можно преобразовать в тензор целых чисел:

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

Единственное, что изменится в данном случае, — функция потерь. В листинге 4.21 мы взяли функцию потерь `categorical_crossentropy`, предполагающую, что метки получены методом кодирования категорий. С целочисленными метками следует использовать функцию `sparse_categorical_crossentropy`:

```
model.compile(optimizer="rmsprop",  
              loss="sparse_categorical_crossentropy",  
              metrics=["accuracy"])
```

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

## 4.2.7. Важность использования достаточно больших промежуточных слоев

Как вы помните, не следует использовать слои, в которых значительно меньше 46 нейронов, потому что результат является 46-мерным. Давайте посмотрим, что получится, если образуется узкое место для информации из-за промежуточных слоев с размерностями намного меньше 46, например четырехмерных.

**Листинг 4.22.** Модель с узким местом для информации

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(4, activation="relu"),
    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))
```

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

**4.2.8. Дальнейшие эксперименты**

Так же как в конце предыдущего примера, я призываю вас провести следующие эксперименты, чтобы развить свою интуицию в выборе решений по конфигурации.

- Попробуйте использовать слои с большим или меньшим числом измерений: 32, 128 и т. д.
- Мы взяли два промежуточных слоя перед последним слоем классификации с функцией активации `softmax`. Теперь попробуйте использовать один промежуточный слой или три.

**4.2.9. Подведение итогов**

Вот какие выводы вы должны сделать из этого примера.

- Если вы пытаетесь классифицировать образцы данных по  $N$  классам, модель должна завершаться слоем `Dense` размера  $N$ .
- В задаче однозначной многоклассовой классификации заключительный слой модели должен иметь функцию активации `softmax`, чтобы 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 моделью, и истинными распределениями целей.
- Метки в многоклассовой классификации можно обрабатывать двумя способами:
  - кодировать их с применением метода кодирования категорий (также известного как прямое кодирование) и использовать функцию потерь `categorical_crossentropy`;
  - кодировать их как целые числа и использовать функцию потерь `sparse_categorical_crossentropy`.
- Когда требуется классифицировать данные по относительно большому количеству категорий, старайтесь предотвратить появление в модели узких мест для информации из-за промежуточных слоев с недостаточно большим количеством измерений.

## 4.3. 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*, заключающаяся в предсказании не дискретной метки, а значения на непрерывной числовой прямой: например, температуры воздуха на завтра по имеющимся метеорологическим данным или 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.

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

Мы попытаемся предсказать медианную цену на дома в пригороде Бостона в середине 1970-х годов по таким данным, как уровень преступности в районе, ставка местного имущественного налога и т. д. Набор данных, который нам предстоит использовать, имеет интересное отличие от примеров, рассматриваемых выше. Он содержит относительно немного образцов данных: всего 506, разбитых на 404 обучающих и 102 контрольных. 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, третьи — между 0 и 100 и т. д.

#### **Листинг 4.23.** Загрузка набора данных для Бостона

```
from tensorflow.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, напомню: это цены середины 1970-х годов и они не учитывают инфляцию.

### **4.3.2. Подготовка данных**

Было бы проблематично передать в нейронную сеть значения, имеющие разные диапазоны. Она, конечно, сможет автоматически адаптироваться к 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, а 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 4.24.** 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, 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.

### 4.3.3. Конструирование модели

Из-за небольшого количества образцов мы возьмем очень маленькую сеть с двумя 64-мерными промежуточными слоями. Вообще, чем меньше обучающих данных, тем скорее наступит переобучение, а использование маленькой модели — один из способов борьбы с ним.

**Листинг 4.25.** Определение модели

```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

Поскольку нам потребуется  
несколько экземпляров одной  
и той же модели, мы определили  
функцию для ее создания

Модель 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`, модель обучилась бы предсказывать только значения из диапазона между 0 и 1. В данном случае, с линейным последним слоем, модель способна предсказывать значения из любого диапазона.

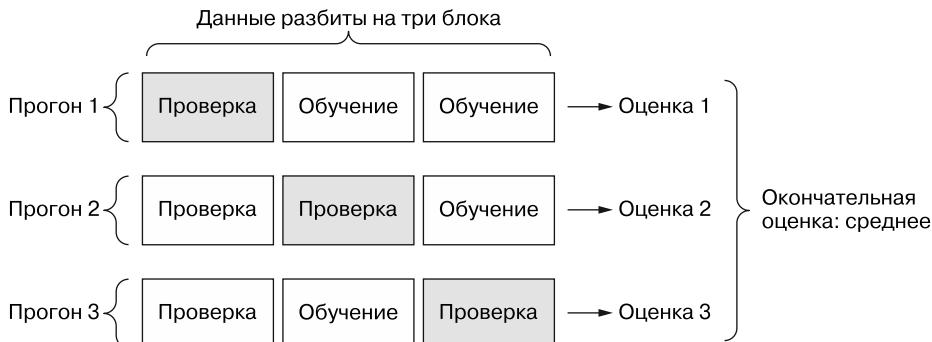
Обратите внимание, что модель компилируется с функцией потерь `mse` — *mean squared error* (среднеквадратичная ошибка), вычисляющей квадрат разности между предсказанными и целевыми значениями. Эта функция широко используется в задачах регрессии.

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 MAE, равное 0,5, в нашей задаче означает, что в среднем прогнозы отклоняются на 500 долларов США.

#### 4.3.4. Оценка решения методом перекрестной проверки по K блокам

Чтобы оценить качество модели 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, поскольку у нас имеется и без того небольшой набор данных, проверочный набор получился бы слишком маленьким (скажем, около 100 образцов). Как следствие, оценки при проверке могут сильно меняться в зависимости от того, какие данные попадут в проверочный и обучающий наборы: иными словами, 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, рис. 4.8).



**Рис. 4.8.** Перекрестная проверка по трем блокам

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.

**Листинг 4.26.** Перекрестная проверка по K блокам

```
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print(f"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]
```

Подготовка проверочных  
данных: данных из блока  
с номером k

```

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()
model.fit(partial_train_data, partial_train_targets,
          epochs=num_epochs, batch_size=16, verbose=0)
val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
all_scores.append(val_mae)

```

Подготовка обучающих данных: данных из остальных блоков

Конструирование модели Keras (уже скомпилированной)

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

Оценка модели по проверочным данным

Выполнив этот код с `num_epochs = 100`, мы получили следующие результаты:

```

>>> all_scores
[2.112449, 3.0801501, 2.6483836, 2.4275346]
>>> np.mean(all_scores)
2.5671294

```

Разные прогоны действительно показывают разные оценки, от 2,1 до 3,1. Средняя (2,6) выглядит более достоверно, чем любая из оценок отдельных прогонов, — в этом главная ценность перекрестной проверки по  $K$  блокам. В данном случае средняя ошибка составила 2600 долларов, что довольно много, если вспомнить, что цены колеблются в диапазоне от 10 000 до 50 000 долларов.

Попробуем увеличить время обучения модели 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 4.27. Сохранение оценки проверки перед каждым прогоном

```

num_epochs = 500
all_mae_histories = []
for i in range(k):
    print(f"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=16, verbose=0)
    mae_history = history.history["val_mae"]
    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.

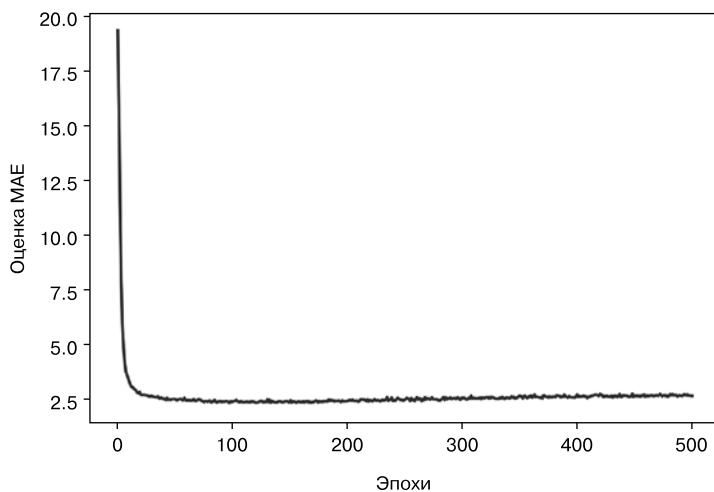
**Листинг 4.28.** Создание истории последовательных средних оценок проверки по К блокам

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

Построим график (рис. 4.9).

**Листинг 4.29.** Формирование графика с оценками проверок

```
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel("Эпохи")
plt.ylabel("Оценка MAE")
plt.show()
```



**Рис. 4.9.** Оценки MAE по эпохам

Из-за проблем с масштабированием может быть немного затруднительно увидеть общую тенденцию: оценка MAE для первых нескольких эпох значительно выше, чем для последующих. Давайте опустим первые десять замеров с масштабом, отличным от масштаба остальной кривой.

**Листинг 4.30.** Формирование графика с оценками проверок за исключением первых десяти замеров

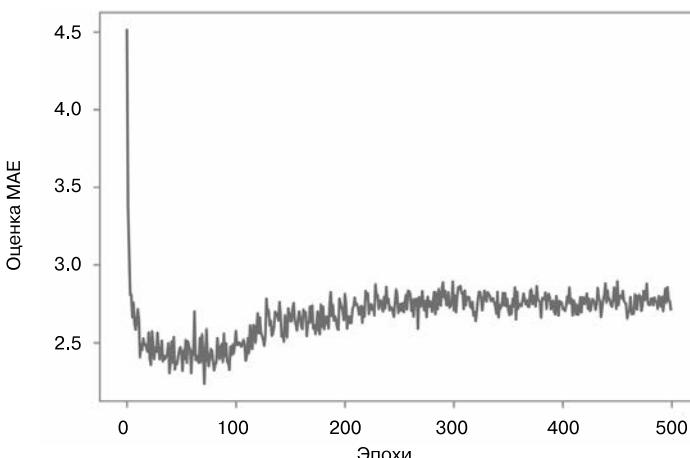
```
truncated_mae_history = average_mae_history[10:]
plt.plot(range(1, len(truncated_mae_history) + 1), truncated_mae_history)
plt.xlabel("Эпохи")
plt.ylabel("Оценка MAE")
plt.show()
```

Как можно увидеть на рис. 4.10, оценка MAE перестает существенно улучшаться после 120–140 эпох (данное число включает десять эпох, которые мы опустили). После этого наступает переобучение.

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

#### Листинг 4.31. Обучение окончательной версии модели

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=130, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```



**Рис. 4.10.** Оценки MAE по эпохам за исключением первых десяти замеров

Вот окончательный результат:

```
>>> test_mae_score
2.4642276763916016
```

Средняя ошибка опустилась ниже 2500 долларов. Это явное улучшение! Как и в двух предыдущих примерах, можете попробовать изменить количество слоев в модели или количество нейронов в каждом слое и посмотреть, удастся ли вам уменьшить ошибку на контрольных данных.

### 4.3.5. Предсказания на новых данных

Вызвав метод `predict()` нашей модели бинарной классификации, мы получили скалярную оценку в диапазоне от 0 до 1 для каждого образца во входной выборке. `predict()` модели многоклассовой классификации вернул распределение вероятностей по всем классам для каждого образца. А `predict()` модели скалярной регрессии возвращает прогноз цены для данного образца в тысячах долларов:

```
>>> predictions = model.predict(test_data)
>>> predictions[0]
array([9.990133], dtype=float32)
```

Таким образом, модель считает, что первый дом в контрольной выборке будет стоить около 10 000 долларов.

### 4.3.6. 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 берется 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  $K$  блокам.
- При небольшом объеме обучающих данных предпочтительнее использовать маленькие модели с небольшим количеством промежуточных слоев (обычно с одним или двумя), чтобы избежать серьезного переобучения.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Наиболее распространенными задачами машинного обучения на векторных данных являются: бинарная классификация, многоклассовая классификация и скалярная регрессия:
  - в разделах «Подведение итогов» выше в этой главе перечисляются наиболее важные выводы, которые вы должны извлечь из примеров решений этих задач;

- при регрессии используются иные функции потерь и метрики, нежели при классификации.
- 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 какой-то момент появляется эффект переобучения, из-за чего падает качество результатов оценки сети на данных, которые она прежде не видела.
- При небольшом объеме обучающих данных используйте небольшие модели 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 небольшом объеме входных данных надежно оценить качество модели поможет метод перекрестной проверки по  $K$  блокам.

# 5

## *Основы машинного обучения*

### **В этой главе**

- ✓ Понимание противоречий между общностью и оптимизацией — фундаментальная проблема машинного обучения.
- ✓ Формальные процедуры оценки моделей машинного обучения.
- ✓ Методы эффективного обучения моделей.
- ✓ Методы достижения лучшего обобщения.

После трех практических примеров в главе 4 у вас должно сложиться начальное понимание, как решаются задачи классификации и регрессии с использованием нейронных сетей. Кроме того, вы собственными глазами увидели главную проблему машинного обучения: переобучение. В этой главе мы формализуем некоторые из новых знаний в прочную концептуальную основу, подчеркнув важность точной оценки модели и баланса между обучением и обобщением.

### **5.1. ОБОБЩЕНИЕ: ЦЕЛЬ МАШИННОГО ОБУЧЕНИЯ**

В трех примерах, представленных в главе 4, — в прогнозировании оценки обзоров фильмов, классификации тем новостей и регрессии цен на жилье — мы делили данные на обучающую, проверочную и контрольную выборки. Мы быстро выяснили причину, почему нельзя оценивать качество моделей по тем же данным, на которых производилось обучение: спустя всего несколько эпох качество

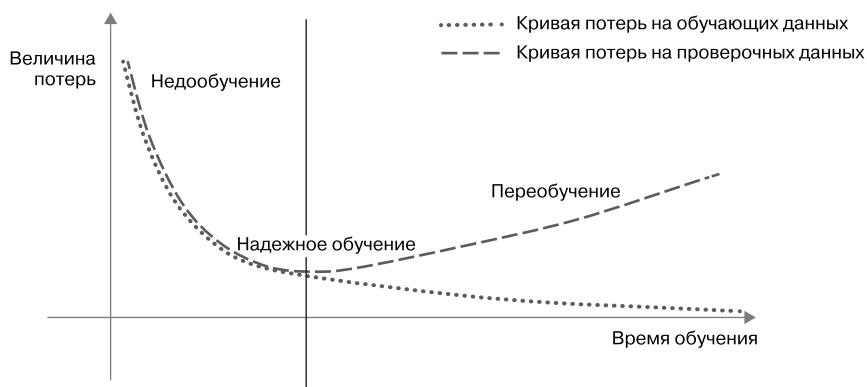
прогнозирования по обучающим данным начинало расходиться с качеством прогнозирования по данным, которые модель прежде не видела. Возникал эффект *переобучения*, и этому эффекту подвержены все модели машинного обучения.

Основной проблемой машинного обучения является противоречие между оптимизацией и общностью. Под *оптимизацией* понимается процесс настройки модели для получения максимального качества на обучающих данных (*обучение в машинном обучении*), а под *общностью* — способность обученной модели давать качественный прогноз по данным, ранее ей незнакомым. Цель игры — добиться высокого уровня общности — но вы не можете ею управлять. Можно только настраивать модель, опираясь на обучающие данные. И если *переусердствовать* в этом, возникнет эффект переобучения и общность пострадает.

Но из-за чего возникает переобучение? Как добиться хорошего уровня общности?

### 5.1.1. Недообучение и переобучение

Во всех трех примерах предыдущей главы качество модели на проверочных данных всегда достигало максимума после небольшого количества эпох и затем начинало снижаться. Это закономерность (рис. 5.1). Далее вы увидите, что она проявляется в моделях любых типов и с любым набором данных.



**Рис. 5.1.** Каноническое поведение модели при переобучении

В начале обучения оптимизация и общность коррелируют: чем ниже потери на обучающих данных, тем они ниже и на контрольных данных. Пока так происходит, принято говорить, что модель *недообучена*: прогресс еще возможен; сеть еще не смоделировала все релевантные шаблоны в обучающих данных. Однако после нескольких итераций на обучающих данных общность перестает

улучшаться, метрики оценки на проверочных данных останавливают свой рост и затем начинают ухудшаться: наступает эффект *переобучения* модели. Другими словами, модель начинает запоминать шаблоны, характерные для обучающих данных, но нетипичные для новых.

Переобучение особенно вероятно, когда исходные данные зашумлены, если в них присутствует элемент неопределенности или они содержат редкие признаки. Давайте рассмотрим конкретные примеры.

### Зашумленные обучающие данные

В реальных наборах данных довольно часто некоторые входные данные оказываютсяискаженными. Например, изображение цифры в наборе MNIST может не содержать ничего, кроме черных пикселей, или выглядеть необычно (рис. 5.2).



**Рис. 5.2.** Некоторые необычные образцы в обучающей выборке в наборе MNIST

Что это за цифры? Я тоже не знаю. Однако все они присутствуют в обучающей выборке в наборе MNIST. Но это полбеды. Что еще хуже, так это наличие совершенно правильных входных данных, которые имеют ошибочные метки (рис. 5.3).



Рис. 5.3. Образцы в обучающей выборке в наборе MNIST, имеющие неверные метки

Если модель постарается учиться такие выбросы, ее способность к обобщению ухудшится (рис. 5.4). Например, рукописная цифра 4, которая очень похожа на 4 с ошибочной меткой на рис. 5.3, может в конечном итоге классифицироваться как 9.

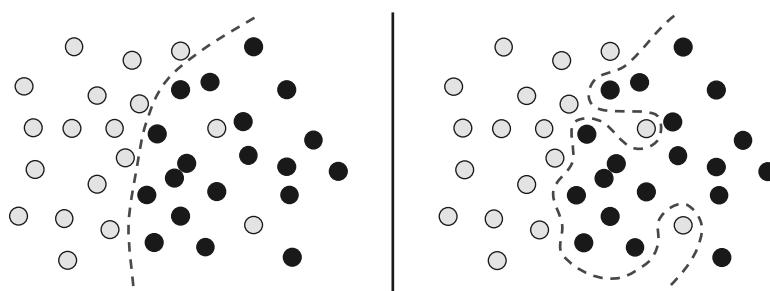


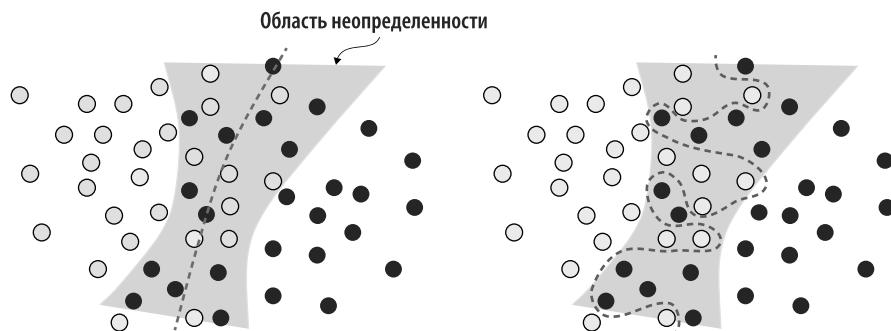
Рис. 5.4. Обобщение выбросов: надежное обучение против переобучения

### Неоднозначные признаки

Не всякий шум в данных возникает из-за неточностей — даже идеально отобранные и аккуратно промаркованные данные могут быть зашумлены, когда имеет место неопределенность и неоднозначность. В задачах классификации часто бывает так, что некоторые области пространства исходных признаков связаны сразу с несколькими классами. Представьте, что вы разрабатываете модель, которая по изображению банана предсказывает — является он незрелым, спелым или гнилым. Эти категории не имеют объективных границ, поэтому одно и то же изображение может быть классифицировано разными людьми как изображение незрелого или спелого банана. Аналогично многие задачи связаны со случайностью. Например, вы могли бы взяться за предсказание дождя по атмосферному давлению, но за одинаковыми измерениями с некоторой вероятностью иногда следует дождливая, а иногда солнечная погода.

Модель могла бы переобучиться — запомнить такие вероятностные данные в обучающей выборке, игнорируя возможность присутствия областей

неопределенности в пространстве признаков (рис. 5.5). На уровне надежного обучения модель способна игнорировать отдельные образцы и видеть картину в целом.



**Рис. 5.5.** Надежно обученная и переобученная модели по-разному видят область неопределенности в пространстве признаков

### Редкие признаки и ложная корреляция

Если вы в своей жизни видели только двух рыжих полосатых кошек и обе они были жутко агрессивными, вы можете сделать вывод, что рыжие полосатые кошки, как правило, агрессивны. Перед вами наглядный пример переобучения: если бы вы встретили большее количество кошек, и не только рыжих, то узнали бы, что характер кошки на самом деле не зависит от ее окраски.

Точно так же модели машинного обучения, натренированные на наборах данных, включающих редкие значения признаков, подвержены переобучению. Если в задаче классификации отзывов в одном из текстов обучающей выборки появится слово *cherimoaya* (черимоя — плод, произрастающий в Андах), а отзыв будет отрицательным, то слабо регуляризованная модель может придать этому слову слишком большой вес — и всегда относить к отрицательным любые новые отзывы, упоминающие черимойю, тогда как объективно в этой ягоде нет ничего плохого<sup>1</sup>.

Важно отметить, что значение признака не обязательно должно встречаться всего несколько раз, чтобы породить ложные корреляции. Представьте слово, которое появляется в ста обучающих образцах, из которых 54 % имеют положительную оценку, а 46 % — отрицательную. Эта разница вполне может быть обусловлена статистической погрешностью, но модель, весьма вероятно, научится

<sup>1</sup> Марк Твен даже назвал черимойю «самым вкусным из известных человечеству фруктом».

использовать данный признак для решения задачи классификации. Это один из самых распространенных источников переобучения.

Вот яркий пример. Возьмем набор MNIST. Создадим новую обучающую выборку, добавив к существующему 784-мерному измерению с фактическими данными такое же 784-мерное измерение с белым шумом, чтобы белый шум занимал половину данных. Для сравнения создадим эквивалентную выборку, добавив 784-мерное измерение с нулями. Добавление бессмысленных признаков никак не влияет на информационное содержание данных: просто в выборке появились измерения, не несущие никакой информации. Эти дополнения никак не повлияют на способность человека различать рукописные цифры.

**Листинг 5.1.** Добавление пустых признаков и признаков с белым шумом в выборку с данными из набора MNIST

```
from tensorflow.keras.datasets import mnist
import numpy as np

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

train_images_with_noise_channels = np.concatenate(
    [train_images, np.random((len(train_images), 784))], axis=1)

train_images_with_zeros_channels = np.concatenate(
    [train_images, np.zeros((len(train_images), 784))], axis=1)
```

Теперь натренируем модель из главы 2 на обеих обучающих выборках.

**Листинг 5.2.** Обучение модели на выборке, включающей пустые признаки и признаки с белым шумом

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model():
    model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

model = get_model()
history_noise = model.fit(
```

```

train_images_with_noise_channels, train_labels,
epochs=10,
batch_size=128,
validation_split=0.2)

model = get_model()
history_zeros = model.fit(
    train_images_with_zeros_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2)

```

И посмотрим, как изменяется точность обеих моделей на проверочной выборке в процессе обучения.

**Листинг 5.3.** Вывод сравнительного графика изменения точности моделей на проверочной выборке

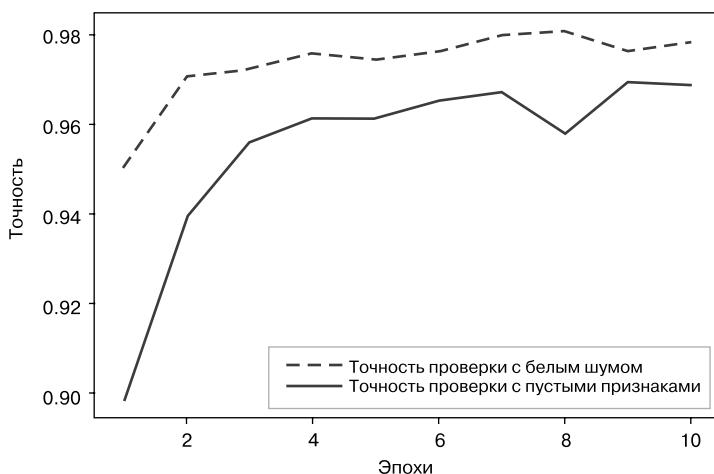
```

import matplotlib.pyplot as plt
val_acc_noise = history_noise.history["val_accuracy"]
val_acc_zeros = history_zeros.history["val_accuracy"]
epochs = range(1, 11)
plt.plot(epochs, val_acc_noise, "b-",
         label="Точность проверки с белым шумом")
plt.plot(epochs, val_acc_zeros, "b--",
         label="Точность проверки с пустыми признаками")
plt.title("Влияние признаков с белым шумом на точность проверки")
plt.xlabel("Эпохи")
plt.ylabel("Точность")
plt.legend()

```

Несмотря на то что обе выборки содержат одну и ту же информацию, точность модели, обученной на выборке с белым шумом, оказалась примерно на один процент ниже (рис. 5.6) — очевидно, что это отставание обусловлено влиянием ложных корреляций. Чем больше каналов с шумом будет добавлено в выборку, тем хуже будет точность.

Зашумленные признаки неуклонно ведут к переобучению. Поэтому, когда нет четкой уверенности в информативности признаков, перед обучением прибегают к *отбору признаков*. Например, ограничение данных из IMDB десятью тысячами самых распространенных слов было упрощенной формой отбора признаков. Типичный способ отобрать признаки — вычислить некоторую оценку полезности для каждого доступного признака, показывающую, насколько он информативен для решаемой задачи (например, тесноту связи между признаком и метками), и оставить только признаки с оценкой выше некоторого порога. Это поможет отфильтровать неинформативные признаки, такие как канал с белым шумом в предыдущем примере.



**Рис. 5.6.** Влияние признаков с белым шумом на точность проверки

### 5.1.2. Природа общности в глубоком обучении

Модели глубокого обучения обладают одним интересным свойством: их можно обучить подстраиваться под что угодно, если они обладают достаточной репрезентативной способностью.

Не верите? Попробуйте перемешать метки в обучающей выборке из набора MNIST и натренировать на ней модель. Несмотря на отсутствие связи между входными данными и перетасованными метками, потери на обучающей выборке снижаются очень хорошо даже в случае относительно небольшой модели. Естественно, потери на проверочной выборке не улучшаются со временем, поскольку отсутствует возможность обобщения.

**Листинг 5.4.** Подстройка модели MNIST под случайно перемешанные метки

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

random_train_labels = train_labels[:]
np.random.shuffle(random_train_labels)

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

```
model.fit(train_images, random_train_labels,  
          epochs=100,  
          batch_size=128,  
          validation_split=0.2)
```

Фактически не обязательно даже использовать данные MNIST — можно просто сгенерировать белый шум и случайные метки. Вы все равно сможете обучить модель под эти данные, если у нее будет достаточно параметров. Она просто запомнит определенные входные данные, подобно словарю в языке Python.

Если это так, то откуда у моделей глубокого обучения может взяться способность к обобщению? Разве они не должны просто выучить связи между обучающими данными и целями, подобно своеобразному словарю `dict`? Разве можно ожидать, что этот словарь будет находить выходные значения для новых ключей на входе?

Оказывается, природа общности в глубоком обучении мало связана с самими моделями глубокого обучения — она намного сильнее связана со структурой информации в реальном мире. Посмотрим, что происходит на самом деле.

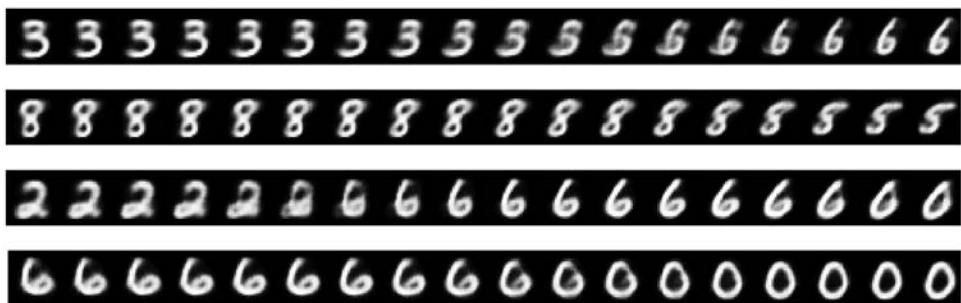
### Гипотеза многообразия

На вход классификатора MNIST подается массив (до предварительной обработки)  $28 \times 28$  целых чисел со значениями от 0 до 255. То есть общее количество возможных входных значений составляет 256 в степени 784 (что намного больше количества атомов во Вселенной). Однако очень немногие из этих входных массивов будут выглядеть как образцы рукописных цифр в MNIST: изображения рукописных цифр занимают лишь малую *подобласть в пространстве* всех возможных массивов  $28 \times 28$  значений `uint8`. Более того, данная область хорошо структурирована: это не просто набор точек, хаотично разбросанных в родительском пространстве.

Прежде всего, область изображений рукописных цифр является *непрерывной*: если взять образец и немного изменить его, он все равно будет распознаваться как та же рукописная цифра. Кроме того, все образцы в этой области *связаны* плавными переходами. То есть для двух случайных цифр А и В из набора MNIST существует последовательность промежуточных изображений, иллюстрирующая такое превращение А в В, что любые две соседние цифры в этой последовательности будут выглядеть очень близкими друг к другу (рис. 5.7). Возможно, рядом с границей, разделяющей два класса, появится несколько неоднозначных изображений, но даже они будут очень похожими на цифры.

С технической точки зрения можно сказать, что рукописные цифры образуют *многообразие* (manifold) в пространстве возможных массивов  $28 \times 28$  значений `uint8`. Звучит мудрено, но сама идея довольно проста. Многообразие — это подобласть меньшей размерности в некотором родительском пространстве, которое

похоже на линейное (евклидово) пространство. Например, гладкая кривая на плоскости — это одномерное многообразие в двумерном пространстве, потому что для каждой точки кривой можно провести касательную (сама кривая может быть аппроксимирована линией в каждой точке). Гладкая поверхность в трехмерном пространстве — это двумерное многообразие. И так далее.



**Рис. 5.7.** Изображения разных цифр через серию шагов можно превратить друг в друга. Это показывает, что пространство рукописных цифр образует «многообразие». Представленные изображения были созданы с использованием кода из главы 12

В более общем плане *гипотеза многообразия* утверждает, что все естественные данные покоятся на многообразии меньшей размерности, находящемся в пространстве большей размерности, где эти данные закодированы. Довольно сильное утверждение о структуре информации во Вселенной. Насколько мне известно, оно верно — и именно поэтому глубокое обучение работает. Оно релевантно для изображений цифр в наборе MNIST, а также для человеческих лиц, деревьев, звуков человеческого голоса и даже для естественного языка.

Из гипотезы многообразия следует, что:

- модели машинного обучения должны подстраиваться только под относительно простые, низкоразмерные и хорошо структурированные подпространства внутри пространства возможных входных данных (скрытых многообразий);
- внутри одного из этих многообразий всегда можно выполнить *интерполяцию* между двумя входами, то есть превратить одно в другое, выполнив последовательность небольших преобразований, все результаты которых принадлежат многообразию.

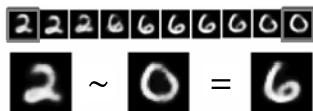
Способность проводить интерполяцию между образцами является ключом к пониманию обобщения в глубоком обучении.

## Интерполяция как ключ к обобщению

Работая с образцами данных, которые можно интерполировать друг в друга, вы в какой-то момент начинаете понимать образцы, которых раньше никогда не видели, сопоставляя их с другими образцами, лежащими поблизости на многообразии. То есть можно прийти к пониманию *всего* пространства, используя ограниченную выборку *образцов*. Для восполнения пробелов можно применить интерполяцию.

Обратите внимание, что интерполяция на скрытом многообразии отличается от линейной интерполяции в родительском пространстве (рис. 5.8). Например, среднее количество пикселей в двух изображениях цифр в наборе MNIST обычно не является допустимой цифрой.

Важно отметить, что глубокое обучение обеспечивает обобщение посредством интерполяции на основе изученного приближения многообразия данных, и было бы ошибкой думать, что интерполяция — *все*, что нужно для обобщения. Это лишь верхушка айсберга. Интерполяция может помочь разобраться только в чем-то очень близком увиденному вами раньше: она дает возможность *локального обобщения*. Самое интересное, что люди постоянно сталкиваются с чем-то совершенно новым — и у них не возникает проблем. Вам не нужно заранее обучаться на бесчисленных примерах каждой предполагаемой ситуации. Любой ваш день отличается от предыдущего, пережитого вами и кем бы то ни было с момента зарождения человечества. Вы можете пробыть неделю в Нью-Йорке, неделю в Шанхае и неделю в Бангалоре — и вам не понадобится тысяча репетиций для жизни там.



Интерполяция на многообразии  
(промежуточная точка  
на скрытом многообразии)



Линейная интерполяция  
(среднее в пространстве  
кодирования)

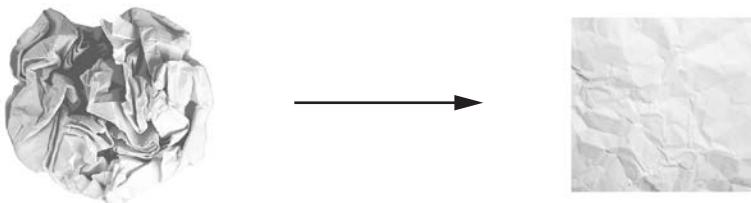
**Рис. 5.8.** Разница между линейной интерполяцией и интерполяцией на скрытом многообразии. Всякий образец из скрытого многообразия изображений цифр является допустимой цифрой, но среднее двух изображений — обычно нет

Люди обладают *развитой способностью обобщать*, основанной на когнитивных механизмах, отличных от интерполяции: абстракции, символическом

моделировании мира, рассуждении, логике, здравом смысле, врожденной подготовленности к жизни, — на всем том, что мы обычно называем *разумом*. В отличие от интуиции и распознавания закономерностей, они в основном имеют интерполяционный характер. Однако и те и другие механизмы важны для интеллекта. Мы еще вернемся к этой теме в главе 14.

### Почему работает глубокое обучение

Помните метафору скомканного листа бумаги из главы 2? Лист бумаги — это двумерное многообразие в трехмерном пространстве (рис. 5.9). Модель глубокого обучения — инструмент для распутывания бумажных шариков (то есть скрытых многообразий).



**Рис. 5.9.** Разглаживание смятого комка исходных данных

Модель глубокого обучения — это, по сути, многомерная кривая, обязательно гладкая и непрерывная (с дополнительными ограничениями на структуру, обусловленными архитектурой модели), раз она должна быть дифференцируемой. И эта кривая подгоняется под точки данных с помощью метода градиентного спуска, плавно и постепенно. По самой своей природе глубокое обучение заключается в том, чтобы взять сложную кривую — многообразие — и постепенно корректировать ее параметры, пока она не будет соответствовать некоторым точкам обучающих данных.

Кривая имеет достаточно параметров, чтобы соответствовать чему угодно — действительно, если позволить модели обучаться достаточно долго, она закончит тем, что просто запомнит обучающие данные и вообще лишится общности. Однако обучающие данные состоят не из изолированных точек, редко разбросанных по основному пространству, — они образуют хорошо структурированное многообразие меньшей размерности во входном пространстве (это и есть гипотеза многообразия). И поскольку подгонка кривой к данным происходит постепенно, то во время обучения найдется промежуточная точка, в которой модель будет близко аппроксимировать естественное множество данных (рис. 5.10).

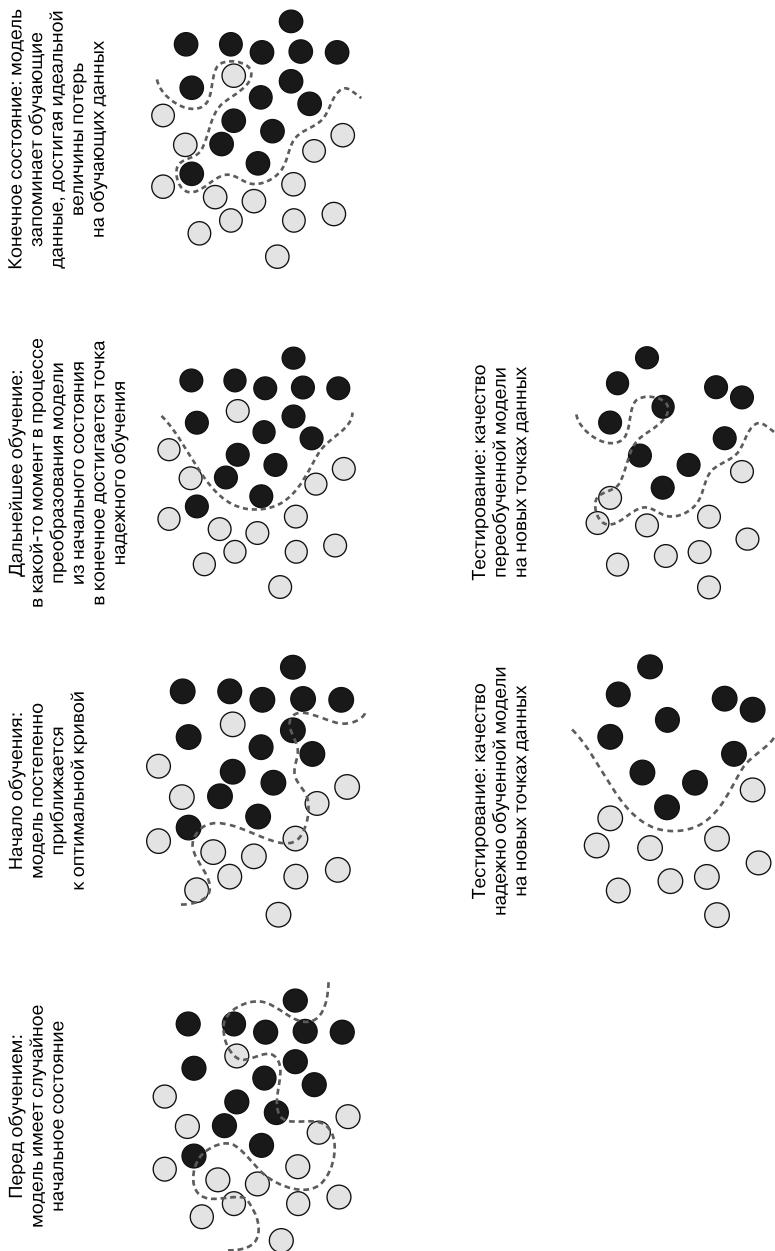


Рис. 5.10. Переход от случайной модели к переобученной через промежуточную точку надежного обучения

Движение по кривой, полученной моделью в этой точке, будет близко к движению по фактическому скрытому многообразию — модель будет способна интерпретировать новые данные, которые прежде не видела, интерполируя их между обучающими входными данными.

Помимо тривиального факта, что у моделей глубокого обучения есть достаточная репрезентативная мощность, они обладают еще несколькими свойствами, которые делают их пригодными для изучения скрытых многообразий:

- модели глубокого обучения реализуют гладкое непрерывное отображение входных данных в выходные. Отображение должно быть гладким и непрерывным, что обусловлено требованием к дифференцируемости (иначе нельзя было бы применить градиентный спуск). Эта гладкость помогает аппроксимировать скрытые многообразия, обладающие теми же свойствами;
- модели глубокого обучения, как правило, имеют структуру, отражающую «форму» информации в обучающих данных (посредством архитектуры). Это особенно верно для моделей обработки изображений (обсуждаемых в главах 8 и 9) и моделей обработки последовательностей (глава 10). В более общем смысле глубокие нейронные сети структурируют изученные представления в иерархии и блоки так, что они перекликаются с организацией естественных данных.

### **Обучающие данные имеют первостепенное значение**

Глубокое обучение действительно хорошо подходит для изучения многообразий, однако способность к обобщениям является скорее следствием естественной структуры данных, а не какого-либо свойства модели. Возможность обобщения появится только тогда, когда данные образуют множество, в котором получится интерполировать точки. Чем информативнее признаки и чем менее они зашумлены, тем проще вам будет обобщать информацию (поскольку пространство исходных данных будет проще и лучше структурировано). Курортирование данных и конструирование признаков чрезвычайно важны для обобщения.

Кроме того, поскольку глубокое обучение — это подгонка кривой, то, чтобы получить надежную модель, *ее необходимо обучать на плотной выборке из входного пространства*. Под плотной выборкой здесь подразумевается плотное покрытие обучающими данными всей входной совокупности (рис. 5.11). Это особенно верно вблизи границ принятия решений. При достаточно плотной выборке появляется возможность оценивать новые входные данные путем интерполяции между предыдущими входными данными без необходимости использовать здравый смысл, абстрактные рассуждения или внешние знания о мире — все то, чего не имеют модели машинного обучения.



**Рис. 5.11.** Чтобы получить модель, способную к точному обобщению, необходима плотная выборка

Поэтому вы всегда должны помнить, что самый надежный способ улучшить модель глубокого обучения — это обучить ее на большом количестве данных или на более точных данных (разумеется, добавление чрезмерно зашумленных или неточных данных повредит общности). Более плотный охват входного многообразия даст модель, обладающую лучшей способностью к обобщению. Не следует ожидать, что модель глубокого обучения будет способна на что-то большее, чем простая интерполяция между обучающими образцами, поэтому старайтесь сделать все возможное, чтобы максимально интерполяцию упростить. В модели глубокого обучения вы найдете только то, что в нее вложите: априорные значения, закодированные в ее архитектуре, и данные, на которых она была обучена.

Когда получить больше данных невозможно, следующим лучшим шагом будет изменение объема информации, хранимого моделью, или добавление ограничений на гладкость кривой модели. Если сеть может позволить себе запомнить только небольшое количество закономерностей или только простые закономерности, то процесс оптимизации заставит ее сосредоточиться на наиболее выделяющихся закономерностях, которые имеют больше шансов на хорошее обобщение. Такой способ борьбы с переобучением называется *регуляризацией*. Мы рассмотрим методы регуляризации в подразделе 5.4.4.

Прежде чем приступить к настройке модели и улучшить ее общность, вы должны найти некоторый способ, который поможет оценить работу модели в данный момент. В следующем разделе вы узнаете, как наблюдать за изменением общности модели во время ее разработки.

## 5.2. ОЦЕНКА МОДЕЛЕЙ МАШИННОГО ОБУЧЕНИЯ

Контролировать можно только то, что доступно для наблюдения. Поскольку цель машинного обучения состоит в создании моделей, успешно обобщающих новые данные, важно иметь возможность надежно оценивать качество обобщения модели. Далее я представлю некоторые способы оценки моделей машинного обучения, большую часть из которых вы уже видели в предыдущих главах.

### 5.2.1. Обучающие, проверочные и контрольные наборы данных

Оценка модели всегда сводится к делению доступных данных на три выборки: обучающую, проверочную и контрольную (или тестовую). Вы тренируете модель на обучающих данных и оцениваете с использованием проверочных. После создания окончательной версии модель тестируется с применением контрольных данных.

Вероятно, вы зададите вопрос: почему бы не использовать только две выборки, обучающую и контрольную? Можно было бы тренировать модель на обучающих 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). В качестве сигнала обратной связи тогда используются проверочные данные. Сама настройка фактически является разновидностью *обучения*: поиском 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 притом, что модель никогда напрямую не обучается на них.

Главная причина этого явления — так называемая *утечка информации*. Каждый раз, настраивая гиперпараметр модели, исходя из качества 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 попадет 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, поэтому для оценки качества модели следует использовать отдельный набор, 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.

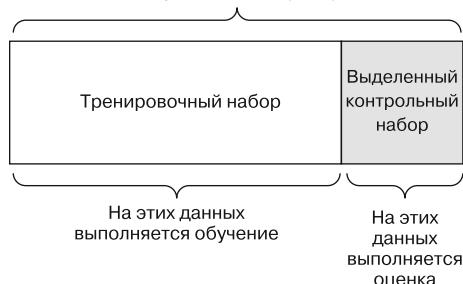
Деление данных на обучающую, проверочную и контрольную выборки может показаться простой задачей. Тем не менее есть ряд продвинутых приемов ее решения, которые могут пригодиться при ограниченном объеме исходных данных. Рассмотрим три классических рецепта оценки: проверку с простым расщеплением выборки (*hold-out validation*), перекрестную проверку по  $K$  блокам (*K-fold validation*) и итерационную проверку 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 выше, 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. 5.12. В листинге 5.5 демонстрируется простейшая реализация этого приема.

Полный объем доступных классифицированных данных



**Рис. 5.12.** Деление данных при использовании проверки с простым расщеплением выборки

**Листинг 5.5.** Проверка с простым расщеплением выборки

```

num_validation_samples = 10000
np.random.shuffle(data)

validation_data = data[:num_validation_samples]
training_data = data[num_validation_samples:]
model = get_model()
model.fit(training_data, ...)
validation_score = model.evaluate(validation_data, ...)

...
model = get_model() ///
model.fit(np.concatenate([training_data,
                           validation_data]), ...)
test_score = model.evaluate(test_data, ...)

```

Перемешивание данных нередко весьма желательно

Формирование проверочной выборки

Формирование обучающей выборки

Обучение модели на обучающих и оценка на проверочных данных

В этой точке можно выполнить корректировку модели, повторно обучить ее, оценить и снова скорректировать

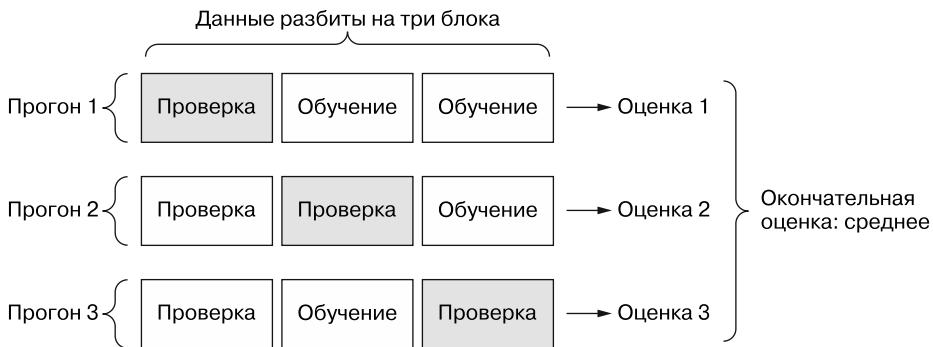
После настройки гиперпараметров часто желательно выполнить обучение окончательной модели на всех данных, не включенных в контрольный набор

Настоящий протокол оценки самый простой, но он страдает одним существенным недостатком: при небольшом объеме доступных данных проверочная и контрольная выборки могут содержать слишком мало образцов, чтобы считаться статистически репрезентативными. Это легко заметить: если разные случайные перестановки данных перед расщеплением дают сильно отличающиеся оценки качества модели, значит, вы столкнулись именно с такой проблемой. Для ее преодоления были разработаны два других подхода — перекрестная проверка по  $K$  блокам и итерационная проверка по  $K$  блокам с перемешиванием. Обсудим их 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. Такой метод может пригодиться, когда качество модели слишком сильно зависит от деления данных на обучающую/контрольную выборки. Подобно проверке с простым расщеплением выборки, этот метод не избавляет от необходимости использовать отдельную проверочную выборку для калибровки модели.

Схематически перекрестная проверка по  $K$  блокам выглядит, как показано на рис. 5.13. В листинге 5.6 отражена простейшая реализация этого приема.



**Рис. 5.13.** Перекрестная проверка по трем блокам

**Листинг 5.6.** Перекрестная проверка по K блокам (обратите внимание, что метки опущены для простоты)

```

k = 3
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)] | Выбор блока данных
    for проверки
        training_data = np.concatenate( | Использование остальных
            data[:num_validation_samples * fold], | данных для обучения.
            data[num_validation_samples * (fold + 1):]) | Обратите внимание,
    model = get_model() | Создание совершенно новой
    model.fit(training_data, ...) | (необученной) модели
    validation_score = model.evaluate(validation_data, ...) | Общая оценка: среднее
    validation_scores.append(validation_score) | оценок по K блокам
validation_score = np.average(validation_scores) | ←
model = get_model()
model.fit(data, ...)
test_score = model.evaluate(test_data, ...)

Обучение окончательной модели
на всех данных, не вошедших
в контрольный набор
  
```

### 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  $P$  — число итераций), что может быть очень затратным.

### 5.2.2. Выбор базового уровня

Кроме различных протоколов оценки, необходимо также иметь базовый уровень, на который можно опереться.

Обучение модели глубокого обучения немного напоминает пуск ракеты, находящейся в параллельном мире, нажатием кнопки. Вы не слышите и не видите, как ракета взлетает. Вы не можете наблюдать многогранный процесс обучения — он протекает в пространстве с тысячами измерений, и даже если спроектировать процесс в привычное трехмерное представление, вы все равно не сможете его интерпретировать. Единственная обратная связь, которая у вас есть, — это ваши проверочные метрики, например высотомер на невидимой ракете.

Очень важно знать, оторвалась ли ракета от поверхности земли в принципе. На какой высоте был произведен пуск? Ваша модель показывает точность 15 % — но как понять, хорошо это или плохо? Прежде чем начать работу с набором данных, всегда нужно выбрать тривиальный базовый уровень, который вы должны превзойти. Перейдя поставленную черту, вы будете знать, что движетесь в правильном направлении: ваша модель действительно использует информацию из входных данных и генерирует обобщающие прогнозы. Таким базовым уровнем может быть результат случайного классификатора или простейшего метода, не имеющего отношения к машинному обучению.

Например, в задаче классификации цифр из набора MNIST простым базовым уровнем мог бы быть случайный классификатор, показывающий точность 0,1 на проверочных данных; в задаче IMDB такой же классификатор даст точность на проверочных данных на уровне 0,5. В задаче с Reuters базовая точность составит примерно 0,18–0,19 из-за несбалансированности классов. При бинарной классификации, когда 90 % образцов принадлежат классу А и 10 % — классу В, классификатор, всегда прогнозирующий А, уже будет иметь точность 0,9 на проверочных данных — вашей целью будет превзойти этот уровень.

Базовый уровень, на который можно сослаться, имеет большое значение, когда вы начинаете работу над задачей, которую прежде никто не решал. Если вы не можете превзойти тривиальное решение, ваша модель бесполезна — может, вы используете неправильную модель или задача вообще не может быть решена с помощью машинного обучения. В таком случае пора вернуться к чертежной доске.

### 5.2.3. Что важно помнить об оценке моделей

Выбирая протокол оценки, всегда помните:

- *о репрезентативности данных* — обучающая и контрольная выборки 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 обучающая выборка будет содержать классы 0–7, а контрольная — только классы 8–9. Данная ошибка может показаться смешной, однако ее совершают слишком часто. По этой причине всегда желательно *перемешивать* данные перед делением на обучающую и контрольную выборки;
- *о направлении оси времени* — пытаясь предсказать будущее по прошлому (например, погоду на завтра, изменение цен на бирже и т. д.), вы *не* должны перемешивать данные перед делением, поскольку это создаст *временную утечку*: ваша модель фактически будет обучаться по данным в будущем. В таких ситуациях всегда нужно следить, чтобы контрольные данные *следовали непосредственно* за обучающими;
- *об избыточности данных* — если некоторые образцы присутствуют в данных в нескольких экземплярах (частое явление в реальном мире), перемешивание и деление данных на обучающую и проверочную выборки приведет к появлению избыточности между ними. По сути, вы будете проводить тестирование на части обучающих данных — а это худшее из зол! Убедитесь, что обучающая и проверочная выборки не пересекаются.

Имея надежный способ оценки качества модели, вы сможете решить главную проблему машинного обучения — найти баланс между оптимизацией и общностью, недообучением и переобучением.

## 5.3. УЛУЧШЕНИЕ КАЧЕСТВА ОБУЧЕНИЯ МОДЕЛИ

Чтобы добиться идеального уровня обучения, сначала нужно переобучить модель. Поскольку заранее не известно, где проходит граница переобучения, чтобы ее обнаружить, нужно ее пересечь. Иными словами, приступая к решению некоторой задачи, в первую очередь сконструируйте модель, которая демонстрирует некоторую способность к обобщению и может переобучиться. Получив такую модель, можно сосредоточиться на совершенствовании ее способности к обобщению, борясь с переобучением.

На этом этапе вы столкнетесь с тремя типичными проблемами:

- обучение не начинается: потери в обучающих данных не уменьшаются;
- обучение начинается, но модель не создает осмысленных обобщений: ей не удается превзойти установленный базовый уровень;
- потери на проверочных данных постепенно снижаются и модели удалось превзойти базовый уровень, но эффект переобучения никак не наступает: это говорит о недообученности модели.

Давайте посмотрим, как решить перечисленные проблемы, чтобы достичь первую важную веху в проекте машинного обучения: получить модель, которая обладает некоторой способностью к обобщению (способностью превзойти тривиальный базовый уровень) и может переобучиться.

### 5.3.1. Настройка основных параметров градиентного спуска

Иногда обучение не начинается или замирает слишком рано. Уровень потерь стоит на месте. Это *всегда* можно преодолеть: помните, что модель можно подстроить даже под случайные данные. Даже если все в вашей задаче лишено всякого смысла, вы *все равно* сможете чему-то обучить модель, пусть даже за счет запоминания обучающих данных.

Когда подобное происходит, корень проблемы почти всегда кроется в настройке процесса градиентного спуска: в выборе оптимизатора, начальных значений весов модели, скорости обучения или размера пакета. Все эти параметры взаимозависимы, поэтому часто достаточно настроить скорость обучения и размер пакета, оставив остальные параметры неизменными.

Давайте рассмотрим конкретный пример: обучим модель MNIST из главы 2 с чрезмерно большой скоростью обучения, равной 1.

#### Листинг 5.7. Обучение модели MNIST с чрезмерно большой скоростью обучения

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1.),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

Модель быстро достигает точности обучения на проверочных данных в диапазоне 30–40 %, но не может превзойти этот уровень. Попробуем уменьшить скорость обучения до более разумного значения `1e-2`.

#### Листинг 5.8. Та же модель с меньшей скоростью

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1e-2),
               loss="sparse_categorical_crossentropy",
               metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

Теперь обучение сдвинулось с мертвой точки.

Если вы оказались в подобной ситуации, попробуйте:

- уменьшить или увеличить скорость обучения. Слишком высокая скорость может привести к чрезмерным изменениям весов, не позволяющим градиентному спуску найти минимум (как в предыдущем примере); слишком же низкая скорость может сделать обучение настолько медленным, что создастся впечатление, будто оно остановилось;
- увеличить размер пакета. Пакет с большим количеством образцов позволит получить более информативные и менее зашумленные градиенты (с меньшей дисперсией).

В конце концов вы найдете конфигурацию, с которой начнется обучение.

### 5.3.2. Использование более удачной архитектуры

Вы сконструировали модель, которая обучается, но по какой-то причине оценки качества на проверочных данных совсем не улучшаются. Они не превышают базового уровня случайного классификатора: модель обучается, но не обобщает. Что происходит?

Это, пожалуй, худшая ситуация в машинном обучении, в которой вы можете оказаться. Она явно сигнализирует: *что-то не так в самом вашем подходе* — и здесь трудно сказать, в чем именно заключается проблема. Вот несколько наводок.

Во-первых, может оказаться, что исходные данные просто не содержат достаточного объема информации для прогнозирования целей: задача в том виде, в котором она сформулирована, неразрешима. Примерно это произошло, когда

мы попытались обучить модель MNIST на наборе с перемешанными метками: модель обучалась нормально, но точность на проверочных данных оставалась на уровне 10 %, потому что с таким набором данных было просто невозможно научиться обобщать.

Возможно также, что для решения поставленной задачи не подходит архитектура модели. Например, в главе 10 вы увидите пример прогнозирования временного ряда, когда плотно связанная архитектура не может превзойти тривиальный базовый случай — а вот более подходящая рекуррентная архитектура оказывается более способной к обобщению. Использование модели, основанной на верных предположениях о проблеме, важно для достижения общности: выбирайте правильные архитектуры.

В следующих главах вы познакомитесь с разными архитектурами, подходящими для работы с разными видами данных — изображениями, текстом, временными рядами и т. д. Всегда следует внимательно изучать передовые архитектуры, применяющиеся для решения задач, аналогичных вашей, — высока вероятность, что вы не первый, кто за нее берется.

### 5.3.3. Увеличение емкости модели

Если вам удалось найти подходящую модель, которая показывает постепенное уменьшение потерь и, кажется, достигает хотя бы некоторого уровня общности, поздравляю: вы почти у цели. Теперь вам нужно найти точку, когда наступает переобучение модели.

Рассмотрим следующую небольшую модель — простую логистическую регрессию, обученную на изображениях MNIST.

#### Листинг 5.9. Простая логистическая регрессия на наборе данных MNIST

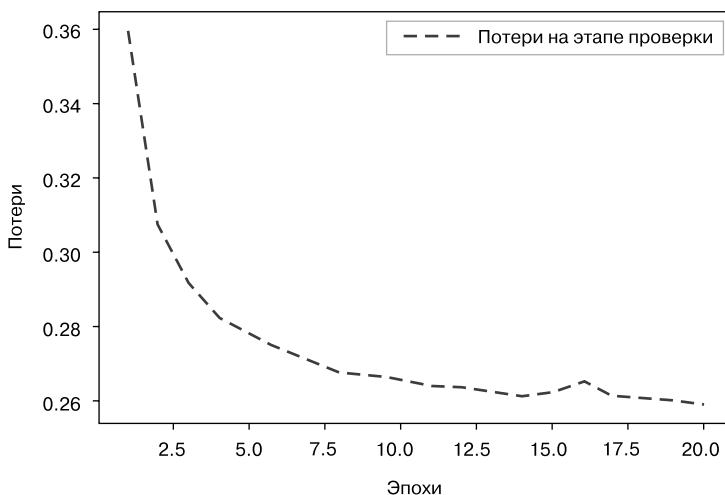
```
model = keras.Sequential([layers.Dense(10, activation="softmax")])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_small_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2)
```

По результатам обучения получилась кривая потерь, изображенная на рис. 5.14:

```
import matplotlib.pyplot as plt
val_loss = history_small_model.history["val_loss"]
epochs = range(1, 21)
plt.plot(epochs, val_loss, "b--",
         label="Потери на этапе проверки")
```

```
plt.title("Эффект недостаточной емкости модели")
plt.xlabel("Эпохи")
plt.ylabel("Потери")
plt.legend()
```

Величина потерь на проверочных данных, кажется, остановилась или очень медленно улучшается — вместо того чтобы пойти вспять. Она достигла значения 0,26 и продолжает колебаться около него. Вы можете продолжать обучение, но эффект переобучения не наступает даже после множества итераций. Скорее всего, в своей карьере вы часто будете сталкиваться с подобными случаями.



**Рис. 5.14.** Эффект недостаточной емкости модели

Помните, что в любой задаче должно наступать переобучение. Равно как проблему неуменьшающихся потерь при обучении, данный вопрос также всегда можно решить. Если возникает ощущение, что переобучение не наступает, скорее всего, проблема связана с недостаточной репрезентативной мощностью вашей модели. Попробуйте сконструировать модель большего размера, с большей емкостью (то есть способную хранить больше информации). Повысить репрезентативную мощность можно, добавив больше слоев, используя более крупные слои (с большим количеством параметров) или типы слоев, лучше подходящие для данной задачи (более удачную архитектуру).

Попробуем обучить более крупную модель, состоящую из двух промежуточных слоев по 96 нейронов в каждом:

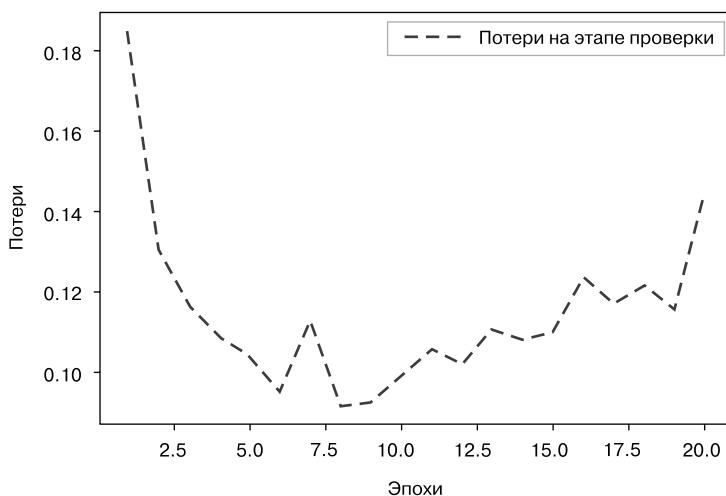
```
model = keras.Sequential([
    layers.Dense(96, activation="relu"),
    layers.Dense(96, activation="relu"),
    layers.Dense(10, activation="softmax"),
])
```

```

model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_large_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2)

```

Кривая потерь на проверочных данных теперь выглядит так, как должна: модель быстро обучается, и через восемь эпох наступает эффект переобучения (рис. 5.15).



**Рис. 5.15.** График кривой потерь для модели с достаточной емкостью

## 5.4. УЛУЧШЕНИЕ ОБЩНОСТИ

Как только модель покажет, что обладает некоторой способностью к обобщению и переобучению, можно переключить внимание на увеличение способности к обобщению.

### 5.4.1. Курирование набора данных

Вы уже знаете, что способность к обобщению в глубоком обучении обусловлена скрытой структурой данных. Если ваши данные позволяют плавно интерполировать между образцами, то вы сможете обучить модель, способную к обобщению. Если данные слишком зашумлены или задача по своей сути дискретная (как, например, сортировка списков), то глубокое обучение вам не поможет. Глубокое обучение — это подгонка кривой, а не волшебство.

Итак, прежде всего важно убедиться, что вы используете подходящий набор данных. Большие затраты сил и средств на сбор данных почти всегда имеют более высокую окупаемость, чем затраты на разработку более совершенной модели.

- Убедитесь в достаточности имеющихся данных. Помните, что вам нужна *плотная выборка* из входного пространства. Чем больше данных, тем лучше модель. Иногда задачи, которые сначала кажутся невозможными, можно решить с помощью более крупного набора данных.
- Минимизируйте ошибки в маркировке — визуализируйте вводимые данные, чтобы наглядно видеть аномалии, и проверяйте и перепроверяйте метки.
- Удалите ошибочные и добавьте недостающие данные (мы поговорим об этом в следующей главе).
- Если у вас слишком много признаков и вы не уверены в том, какие из них действительно полезны, выполните процедуру выбора признаков.

Особенно важным способом улучшить обобщение данных является *конструирование признаков*. Для большинства задач машинного обучения это ключевой ингредиент успеха. Давайте рассмотрим его.

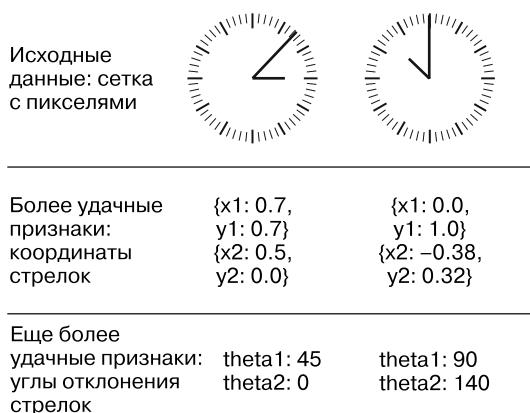
## 5.4.2. Конструирование признаков

*Конструирование признаков* — это процесс использования ваших собственных знаний о данных и алгоритме машинного обучения (в нашем 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).

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 сделать гораздо более удачные входные признаки для алгоритма машинного обучения: например, можно написать пять строк кода на Python, которые проследуют по черным пикселям стрелок и вернут координаты ( $x$ ,  $y$ ) конца каждой стрелки. Тогда простой алгоритм машинного обучения сможет научиться связывать эти координаты с соответствующим временем дня.



**Рис. 5.16.** Конструирование признаков для чтения времени с изображения циферблата часов

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 даже 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. Например, до того, как нейронные сети достигли в этом 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 пикселей 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 работать над задачей 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.

### 5.4.3. Ранняя остановка

В глубоком обучении принято использовать модели с чрезмерным количеством параметров: они имеют гораздо больше степеней свободы, чем необходимо для аппроксимации скрытого многообразия данных. Это не является проблемой, потому что *модели глубокого обучения никогда не обучаются до предела*. Обучение до предела означало бы полное отсутствие способности к обобщению. Вы всегда будете останавливать обучение задолго до достижения минимально возможного уровня потерь на обучающих данных.

Поиск в процессе обучения точки, в которой достигается наибольший уровень общности, — точной границы между недообучением и переобучением — является одним из наиболее эффективных способов улучшить общность.

В примерах из предыдущей главы мы обучали наши модели дольше, чем необходимо, чтобы определить номер эпохи, после которой модель показывает наименьшие потери на проверочных данных, а затем повторно обучали новую модель в течение именно этого количества эпох. Это стандартный подход, но он требует избыточной работы, которая иногда может обходиться дорого. Естественно, можно просто сохранять модель в конце каждой эпохи и, определив лучшую из них, взять соответствующую. В Keras для этого обычно используется обратный вызов `EarlyStopping`, в котором можно прервать обучение, как только показатели качества на проверочных данных перестанут улучшаться, запомнив при этом лучшее состояние модели. Я покажу, как применять обратные вызовы, в главе 7.

### 5.4.4. Регуляризация модели

*Методы регуляризации* — это набор приемов, мешающих модели достичь идеального соответствия обучающим данным, чтобы улучшить ее качество на проверочных данных. Эти приемы называются регуляризацией, потому что позволяют сделать модель более простой, более регулярной, более обобщенной, а кривую потерь — более гладкой. То есть регуляризованная модель получается менее ориентированной на обучающую выборку и имеет лучшую способность к обобщению за счет более точной аппроксимации скрытого многообразия данных.

Имейте в виду, что регуляризация модели — это процесс, в котором всегда следует руководствоваться точной процедурой оценки. Вы сможете достичь высокого уровня общности, только если сможете его измерить.

Давайте рассмотрим некоторые распространенные методы регуляризации и применим их на практике, чтобы улучшить модель классификации отзывов к фильмам из главы 4.

## Уменьшение размера сети

Вы уже знаете, что слишком маленькие модели не подвержены переобучению. Самый простой способ предотвратить переобучение — уменьшить размер модели, иными словами, количество изучаемых параметров в модели (определяется количеством слоев и количеством нейронов в каждом слое). Модель с ограниченными ресурсами не сможет просто запомнить обучающие данные, поэтому для минимизации потерь ей придется прибегнуть к изучению сжатых представлений, обладающих прогнозирующей способностью в отношении целей, — это нам как раз и требуется. В то же время модель должна иметь достаточное количество параметров, чтобы не возник эффект недообучения: помните, 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 числа слоев или правильного размера каждого слоя. Вам придется оценить множество разных архитектур (на вашей проверочной (но не на контрольной!) выборке, конечно), 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; затем следует постепенно увеличивать размеры слоев и их количество, пока не произойдет увеличение потерь на проверочных данных.

Давайте опробуем этот подход на сети, выполняющей классификацию отзывов к фильмам. В листинге 5.10 представлена исходная модель.

### Листинг 5.10. Исходная модель

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

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

train_data = vectorize_sequences(train_data)
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
```

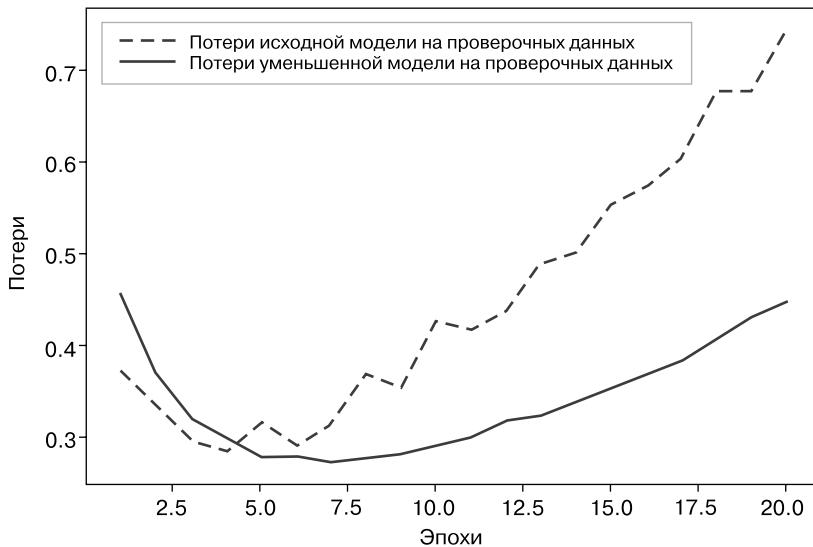
```
metrics=["accuracy"])
history_original = model.fit(train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

Теперь попробуем заменить ее меньшей моделью.

#### Листинг 5.11. Версия модели с меньшей емкостью

```
model = keras.Sequential([
    layers.Dense(4, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
    loss="binary_crossentropy",
    metrics=["accuracy"])
history_smaller_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

На рис. 5.17 для сравнения показаны графики потерь на проверочных данных для исходной модели и ее уменьшенной версии.



**Рис. 5.17.** Сравнение оригинальной и уменьшенной моделей классификации отзывов в наборе данных IMDB

Как видите, эффект переобучения уменьшенной модели возникает позже, чем исходной (после шести эпох, а не четырех), и после начала переобучения ее качество ухудшается более плавно.

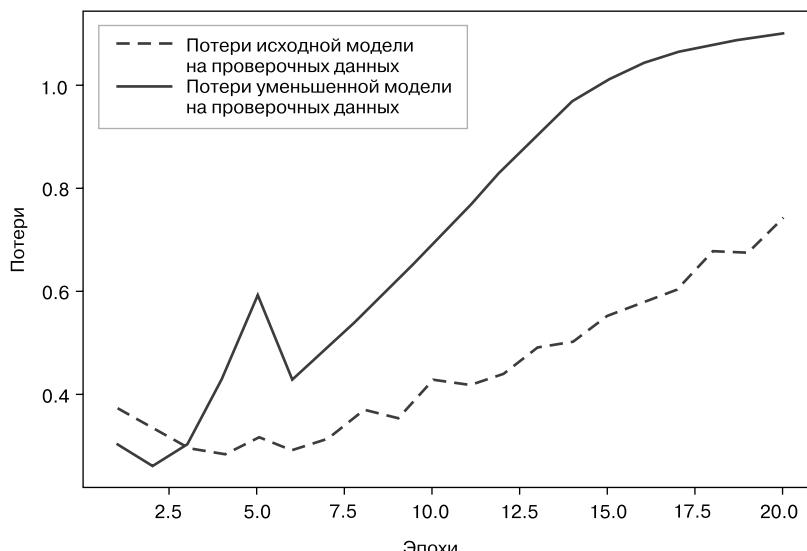
Теперь для контраста добавим сеть с большей емкостью — намного большей, чем необходимо для данной задачи.

В мире машинного обучения принято использовать модели, где количество параметров значительно превышает пространство признаков, которое они пытаются изучить, однако такие модели имеют слишком большую способность к запоминанию. Если модель слишком большая, она почти сразу же начнет переобучаться, а ее кривая потерь на проверочных данных будет выглядеть прерывистой с большой дисперсией (впрочем, большая дисперсия также может говорить о ненадежности процесса проверки, например если проверочная выборка слишком мала).

#### Листинг 5.12. Версия модели с намного большей емкостью

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(512, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_larger_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

На рис. 5.18 для сравнения приводятся графики потерь на проверочных данных для увеличенной и исходной моделей.



**Рис. 5.18.** Сравнение оригинальной и увеличенной моделей классификации отзывов в наборе данных IMDB

Переобучение увеличенной модели наступает почти сразу же, после одной эпохи, и имеет намного более серьезные последствия. Кривая потерь на проверочных данных выглядит слишком искаженной. Большая модель быстро достигает нулевых потерь на обучающих данных. Чем больше емкость модели, тем быстрее она обучается представлять данные (что приводит к очень низким потерям на обучающих данных), но она более восприимчива к переобучению (что дает большую разность между потерями на обучающих и проверочных данных).

## 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 на основе нейронных сетей: для одних и тех же исходных условий — обучающей выборки 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 модели путем ограничения значений ее весовых коэффициентов, что делает их распределение более *равномерным*. Этот прием называется *регуляризацией весов*. Он реализуется добавлением в функцию потерь *штрафа* за увеличение весов и имеет две разновидности, такие как:

- *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 есть 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 5.13).

### Листинг 5.13. Добавление L2-регуляризации весов в модель

```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
```

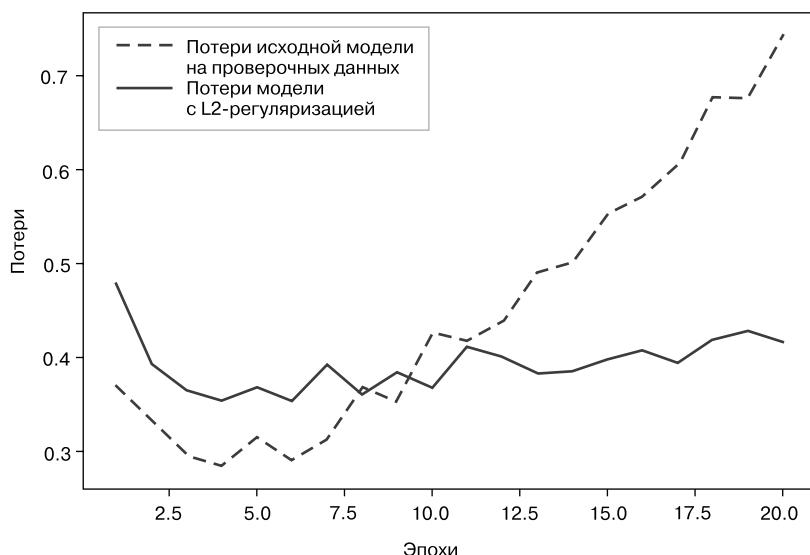
```

        kernel_regularizer=regularizers.l2(0.002),
        activation="relu"),
layers.Dense(16,
            kernel_regularizer=regularizers.l2(0.002),
            activation="relu"),
layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_l2_reg = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)

```

`l2(0.002)` означает, что каждый коэффициент в матрице весов слоя будет добавлять  $0.002 * \text{weight\_coefficient\_value} ** 2$  в общее значение потерь сети. Обратите внимание, что штраф добавляется только на этапе обучения, поэтому величина потерь сети на этапе обучения будет намного выше, чем на этапе контроля.

На рис. 5.19 показано влияние L2-регуляризации. Как видите, модель с L2-регуляризацией намного устойчивее к переобучению, чем исходная, даже притом, что обе модели имеют одинаковое количество параметров.



**Рис. 5.19.** Влияние L2-регуляризации весов на величину потерь на проверочных данных

Вместо L2-регуляризации можно также использовать следующие регуляризаторы, входящие в состав Keras.

#### Листинг 5.14. Разные регуляризаторы, доступные в Keras

```
from tensorflow.keras import regularizers
regularizers.l1(0.001)           ← L1-регуляризация
regularizers.l1_l2(l1=0.001, l2=0.001) ← Объединенная
                                         L1- и L2-регуляризация
```

Обратите внимание, что регуляризация весов чаще применяется в небольших моделях глубокого обучения. Большие модели настолько чрезмерно параметризованы, что наложение ограничений на значения весов не оказывает большого влияния на способность модели к обобщению. В этих случаях предпочтительнее применять другой метод регуляризации — прореживание.

#### Добавление прореживания

*Прореживание* (dropout) — один из наиболее эффективных и распространенных приемов регуляризации нейронных сетей, разработанный Джоном Хинтоном и его студентами в Университете Торонто. Прореживание, которое применяется к слову, заключается в *удалении* (присваивании нуля) случайно выбираемым признакам на этапе обучения. Представьте, что в процессе обучения некоторый слой для некоторого входного образца в нормальной ситуации возвращает вектор [0.2, 0.5, 1.3, 0.8, 1.1]. После 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 `layer_output`, полученную на выходе слоя, 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 0,5, потому что прежде была отброшена половина признаков):

```
layer_output *= 0.5 ← На этапе тестирования
```

Обратите внимание, 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. 5.20):

```
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

0.3	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.2	1.9	0.3	1.2
0.7	0.5	1.0	0.0

Прореживание 50 %

0.0	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.0	1.9	0.3	0.0
0.7	0.0	0.0	0.0

\* 2

**Рис. 5.20.** Прореживание применяется к матрице активации на этапе обучения с масштабированием на этом же этапе. На этапе тестирования матрица активации не изменяется

Этот прием может показаться странным и необоснованным. Каким образом он поможет справиться с переобучением? По словам Хинтона, основой для данного 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 сговора клиента с сотрудником банка. Это навело меня на мысль, что удаление случайно выбранного подмножества нейронов из каждого примера может помочь предотвратить заговор модели с исходными данными и тем самым ослабить эффект переобучения». Иными словами, основная идея заключается в введении шума в выходные значения, способного разбить случайно складывающиеся, не имеющие большого значения шаблоны (Хинтон называет их *заговорами*), которые модель начинает запоминать в отсутствие шума.

В Keras добавить прореживание можно введением в модель слоя `Dropout`, который обрабатывает результаты работы слоя, стоящего непосредственно перед ним. Давайте добавим два слоя `Dropout` в модель IMDB и посмотрим, как это повлияет на эффект переобучения

#### Листинг 5.15. Добавление прореживания в модель IMDB

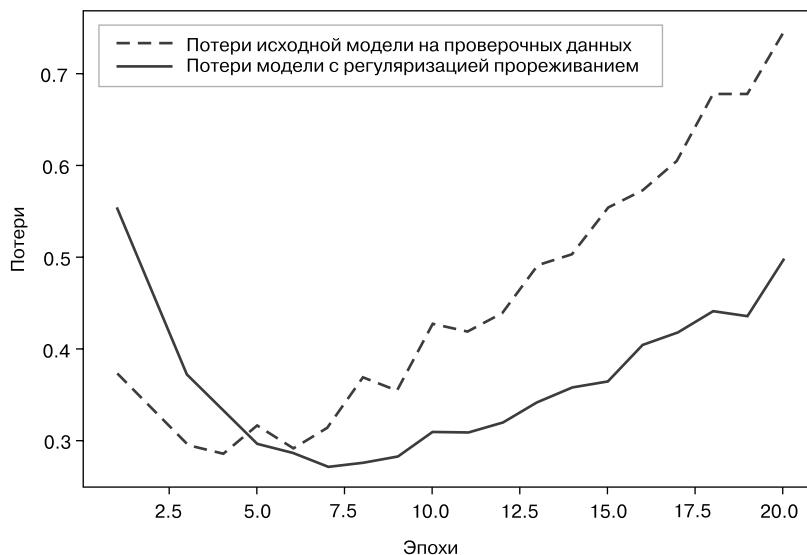
```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
```

```

        layers.Dense(16, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(1, activation="sigmoid")
    ])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_dropout = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)

```

На рис. 5.21 показаны графики с результатами. И снова в сравнении с исходной моделью наблюдается улучшение.



**Рис. 5.21.** Влияние прореживания на величину потерь на проверочных данных

Таким образом, наиболее распространенные способы ослабления проблемы переобучения нейронных сетей следующие:

- увеличить объем обучающих данных и их качество;
- выбрать наиболее информативные признаки;
- уменьшить емкость сети;
- добавить регуляризацию весов (при использовании небольших моделей);
- добавить прореживание.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Цель модели машинного обучения — *обобщение*: обеспечить максимально точную интерпретацию ранее не встречавшихся ей данных. Это сложнее, чем кажется.
- Глубокая нейронная сеть учится обобщать, обучаясь *интерполировать* переходы между обучающими образцами, — можно сказать, что обученная модель изучила «скрытое многообразие» обучающих данных. Вот почему модели глубокого обучения могут более или менее успешно интерпретировать только те данные, которые близки к увиденным ими во время обучения.
- Фундаментальная проблема машинного обучения — *противоречие между оптимизацией и общностью*: чтобы достичь общности, необходимо сначала добиться хорошей подгонки модели под обучающие данные, но через какое-то время улучшение подгонки модели под обучающие данные неизбежно начнет вредить ее общности. Все передовые приемы глубокого обучения направлены на преодоление этого противоречия.
- Способность моделей глубокого обучения к обобщению обусловлена умением аппроксимировать *скрытое многообразие* их данных и за счет этого интерпретировать новые входные данные, применяя интерполяцию.
- Очень важно иметь возможность точно оценить обобщающую способность модели во время ее разработки. Для этого существует множество методов оценки, от простой проверки с расщеплением выборки до перекрестной проверки по  $K$  блокам и перекрестной проверки по  $K$  блокам с перемешиванием. Не забывайте всегда создавать отдельную контрольную выборку для окончательной оценки модели и не используйте ее для проверки в процессе обучения, чтобы исключить утечку информации из контрольных данных в вашу модель.
- Начиная работать над моделью, сначала создайте такую, которая обладает хоть какой-то способностью к обобщению и может переобучаться. В этом вам помогут: настройка скорости обучения и размера пакета, использование более подходящих архитектур, увеличение емкости модели или просто более длительное обучение.
- Получив модель, способную переобучаться, переключитесь на улучшение ее способности к обобщению за счет *регуляризации*. Попробуйте уменьшить емкость модели, добавить регуляризацию весов или прореживание, используйте раннюю остановку. И естественно, не забывайте о данных: чем больше набор данных и чем они качественнее, тем выше будет способность модели к обобщению.



# *Обобщенный процесс машинного обучения*

## **В этой главе**

- ✓ Этапы определения задачи машинного обучения.
- ✓ Этапы разработки действующей модели.
- ✓ Этапы развертывания модели в промышленном окружении и ее обслуживание.

В предыдущих примерах предполагалось, что у нас есть готовый маркированный набор данных и возможность немедленно начать обучение модели. В реальном мире так бывает очень редко. Вам придется начинать не с создания набора данных, а с определения задачи.

Представьте, что вы открываете свой консалтинговый центр по машинному обучению. Вы регистрируетесь, создаете яркий сайт, уведомляете потенциальных клиентов — и вам начинают поступать проекты:

- персонализированный механизм поиска по фотографиям в социальной сети: вы вводите слово «свадьба» — и получаете все фотографии, сделанные на свадьбах, без необходимости добавлять метки вручную;
- определение спама и оскорбительных выражений в сообщениях нового чат-приложения;
- системы подбора музыкальных рекомендаций для слушателей интернет-радио;

- выявление мошеннических действий с кредитными картами на веб-сайте электронной коммерции;
- прогнозирование процента переходов по рекламным ссылкам, чтобы решить, какое объявление показывать тому или иному пользователю в данный момент;
- выявление бракованных изделий на конвейерной ленте линии по производству печенья;
- использование спутниковых изображений для предсказания местонахождения еще не обнаруженных археологических памятников.

### **ЗАМЕЧАНИЕ ПО ЭТИКЕ**

Иногда вам могут поступать проекты, сомнительные с этической точки зрения, например «создание модели ИИ, оценивающей надежность человека по его портрету». Во-первых, сомнительна сама обоснованность такого проекта: непонятно, как надежность может отразиться на лице. Во-вторых, подобная задача открывает двери всевозможным этическим проблемам. Подбор данных для этой задачи равносителен фиксации предубеждений людей, которые будут добавлять метки к изображениям. Модели, обученные на таких данных, перенесут предубеждения в черный ящик алгоритма, придавая им тонкий налет легитимности. В таком технологически малограмотном обществе, как наше, вердикт «алгоритм ИИ сказал, что этому человеку нельзя доверять», как ни странно, будет казаться более весомым и объективным, чем «Джон Смит сказал, что этому человеку нельзя доверять», — несмотря на то что первый представляет лишь усвоенное приближение к последнему. Ваша модель будет обелять и распространять наихудшие аспекты человеческих суждений с негативными последствиями для реальных людей.

Технологии не бывают нейтральными. Если ваша работа имеет какое-либо влияние на мир, это влияние имеет моральную сторону: технические решения являются также этическим выбором. Всегда обдумывайте, какие ценности вы хотите поддерживать в своей работе.

Было бы очень удобно иметь возможность импортировать корректный набор данных из `keras.datasets` и попробовать некоторые модели глубокого обучения. К сожалению, в реальной жизни вам придется начинать с нуля.

В этой главе вы познакомитесь с универсальным пошаговым планом, который можно использовать для решения любых задач машинного обучения — в том числе перечисленных в списке выше. Этот шаблон, объединив и обобщив все, что вы узнали в главах 4 и 5, даст вам дополнительную практику, которая поможет закрепить сведения следующих глав.

Универсальный процесс машинного обучения состоит из трех частей.

1. *Определение задачи* — изучите предметную область и бизнес-логику, лежащую в основе того, о чем просит заказчик. Выполните сбор и первичный анализ данных и выберите критерий оценки успеха в решении задачи.
2. *Разработка модели* — подготовьте данные для передачи в модель машинного обучения; выберите протокол оценки модели и простой базовый уровень, который нужно превзойти; обучите первую модель, обладающую способностью к обобщению и переобучению, а затем настраивайте ее и применяйте регуляризацию, пока не будет достигнуто максимально возможное качество обобщения.
3. *Разворачивание модели* — представьте результаты заинтересованным сторонам; перенесите модель на веб-сервер, в мобильное приложение, веб-страницу или встроенное устройство; наблюдайте за качеством работы модели в реальном времени и начинайте сбор данных, которые вам понадобятся при строительстве модели следующего поколения.

А теперь приступим.

## 6.1. ОПРЕДЕЛЕНИЕ ЗАДАЧИ

Вы не сможете добиться хороших результатов без глубокого понимания предложенной задачи. Почему ваш клиент пытается решить именно ее? Какие выгоды он извлечет из этого — как будет использоваться ваша модель и как она впишется в бизнес-процессы клиента? Какие данные имеются в наличии или могут быть собраны? Какую задачу машинного обучения можно сравнить с бизнес-задачей?

### 6.1.1. Формулировка задачи

Формулировка задачи машинного обучения обычно требует детального обсуждения с заинтересованными сторонами. Вот вопросы, которые вы должны держать в голове.

- Какой вид будут иметь входные данные? Что требуется предсказать? Вы сможете обучить сеть предполагать что-либо только при наличии обучающих данных: например, обучить сеть определять оценку в отзывах к фильмам можно, если имеются отзывы и соответствующие аннотации. То есть доступность данных на данном этапе является ограничивающим фактором. Во многих случаях вам придется прибегать к сбору и аннотированию новых наборов данных самостоятельно (о чем мы поговорим в следующем разделе).
- К какому типу относится задача, стоящая перед вами? Бинарная классификация? Многоклассовая классификация? Скалярная регрессия? Векторная

регрессия? Многоклассовая многозначная (нечеткая) классификация? Сегментация изображения? Определение рейтинга? Что-то иное, например кластеризация, генерация или обучение с подкреплением? В некоторых случаях может оказаться, что машинное обучение не лучший способ обработки данных и следует использовать что-то еще, например старый добрый статистический анализ:

- поисковая система фотографий — это задача многоклассовой многозначной классификации;
  - определение спама — задача бинарной классификации. А если выделить в отдельный класс «оскорбительные выражения» — понадобится тернарная классификация;
  - задачу подбора музыкальных рекомендаций, как оказывается, лучше решать не с помощью глубокого обучения, а методом матричной факторизации (коллаборативной фильтрации);
  - выявление мошеннических действий с кредитными картами — это задача бинарной классификации;
  - прогнозирование процента переходов по рекламным ссылкам — скалярной регрессии;
  - выявление бракованных изделий на линии по производству печенья — задача бинарной классификации, но для этого также потребуется модель обнаружения объектов, которая на первом этапе будет правильно идентифицировать печенье в изображениях ( обратите внимание, что набор методов машинного обучения, известный как «обнаружение отклонений», для этой ситуации не подходит!);
  - поиск новых археологических памятников по спутниковым снимкам — это задача ранжирования изображений по сходству: получить новые изображения, похожие на известные местонахождения археологических памятников.
- Как выглядят существующие решения? Возможно, у вашего клиента уже есть созданный вручную алгоритм, выполняющий фильтрацию спама или выявляющий мошенничество с кредитными картами с использованием множества вложенных операторов `if`. Может, в настоящее время задача решается человеком, который, например, наблюдает за конвейерной лентой на заводе по производству печенья и вручную удаляет бракованные изделия или составляет списки воспроизведения с рекомендациями для рассылки пользователям, которым понравился конкретный исполнитель. Вы должны убедиться, что понимаете, какие системы уже существуют и как они работают.
  - Есть ли какие-то особые ограничения, с которыми вам придется столкнуться? Например, может выясниться, что приложение, для которого нужно создать систему обнаружения спама, в обязательном порядке шифрует сообщения, поэтому разрабатываемая модель должна функционировать на телефоне

конечного пользователя и обучаться на внешнем наборе данных. Возможно, модель выявления бракованных печений имеет такие ограничения по задержкам, что ее придется запускать на встроенном устройстве на заводе, а не на удаленном сервере. Вы должны знать и понимать все условия, в которых придется работать вашей модели.

После проведения исследований хорошо иметь более или менее полное представление о том, какими должны быть входные данные и целевые значения и какой тип задачи машинного обучения соответствует вашей проблеме. Не забывайте о гипотезах, которые выдвигаются на этом этапе:

- 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 идеи, 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.

### **6.1.2. Сбор данных**

Определив природу задачи и узнав, какими должны быть входные данные и цели, можно приступать к сбору данных — наиболее сложной, трудоемкой и дорогостоящей части большинства проектов машинного обучения.

- Для поисковой системы фотографий сначала нужно сформировать набор меток, представляющих нужные вам классы, — на выбор у вас есть 10 000 общих категорий изображений. Затем необходимо вручную пометить сотни тысяч загруженных пользователями изображений метками из этого набора.
- Для определения спама в чат-приложении обучить модель на его содержимом не получится — пользовательские чаты используют сквозное шифрование. Вам потребуется получить доступ кциальному набору данных из десятков тысяч неотфильтрованных сообщений в социальных сетях и вручную пометить их как спам, оскорбительные или допустимые.
- Систему подбора музыкальных рекомендаций можно реализовать на основе лайков, оставляемых вашими пользователями. Никаких новых данных собирать не нужно. Аналогично для прогнозирования процента переходов по рекламным ссылкам можно использовать информацию о переходах по прошлым объявлениям за последние годы.

- Для создания модели отбраковки печенья вам потребуется установить камеры над конвейерными лентами и собрать десятки тысяч изображений; затем кто-то должен будет вручную снабдить эти изображения метками. Люди, занимающиеся отбраковкой, в настоящее время работают на фабрике печенья, но сама по себе эта задача не очень сложная и у вас должно получиться обучить этому же своих сотрудников.
- Для решения задачи анализа спутниковых снимков необходимо, чтобы группа археологов составила перечень существующих археологических объектов, представляющих интерес. Для каждого объекта вы должны будете найти спутниковые снимки, сделанные в различных погодных условиях. Чтобы получить хорошую модель, вам понадобятся тысячи различных местоположений.

В главе 5 вы узнали, что способность модели к обобщению почти полностью зависит от свойств обучающих данных — от количества образцов данных, точности меток, качества признаков. Хороший набор данных — это актив, в который стоит вкладывать время и силы. Если у вас есть дополнительные 50 часов, которые можно потратить на проект, то лучше отдать их на сбор дополнительных данных, а не на поиск улучшений в моделировании.

Идея о большей важности данных по сравнению с алгоритмами наиболее ярко была представлена в статье *The Unreasonable Effectiveness of Data*, написанной исследователями из Google в 2009 году (название является ссылкой к известной статье *The Unreasonable Effectiveness of Mathematics in the Natural Sciences*, написанной Юджином Вигнером в 1960 году). Статья появилась еще до того, как глубокое обучение стало популярным, но, что примечательно, развитие глубокого обучения только увеличило важность данных.

В случае обучения с учителем после сбора входных данных (например, изображений) вам потребуется снабдить их метками (например, добавить теги к изображениям) — целевыми значениями, которые ваша модель должна научиться предсказывать. Иногда метки можно получить автоматически, например для задачи подбора музыкальных рекомендаций или прогнозирования процента переходов по рекламным объявлениям. Но чаще это приходится делать вручную — весьма трудоемкий процесс.

## **Вложения в инфраструктуру маркировки данных**

Процесс маркировки данных определяет качество целевых значений, которые, в свою очередь, определяют качество модели. Внимательно изучите доступные вам варианты такие, как:

- самостоятельная маркировка данных;
- маркировка с привлечением краудсорсинговой платформы (такой как Mechanical Turk);
- помочь специализированной компании по маркировке данных.

Аутсорсинг (маркировка сторонними исполнителями) может помочь сэкономить ваше время и деньги, но лишает вас контроля. Использование такой платформы, как Mechanical Turk, вероятно, обойдется еще дешевле, но качество маркировки может оказаться довольно низким.

Оценивая варианты, учитывайте имеющиеся ограничения.

- Должны ли быть экспертами в предметной области те, кто будет заниматься маркировкой данных, или это под силу любому? Отбором изображений для задачи классификации кошек и собак может заниматься любой, но для определения пород собак нужны специальные знания; маркировка же, к примеру, компьютерных томограмм переломов костей и вовсе требует степени доктора медицины.
- Если маркировка данных требует специальных знаний, то сможете ли вы обучить других людей? Если нет, то как привлечь к этой работе экспертов?
- Понимаете ли вы сами, как эксперты маркируют данные? Если нет, то вам придется рассматривать свой набор данных как черный ящик и у вас не будет возможности проектирования признаков вручную — это некритично, но может стать ограничивающим фактором.

Если вы решите маркировать данные самостоятельно, то спросите себя, какое программное обеспечение будете использовать для записи меток. Возможно, вам придется его разработать самим. Эффективное программное обеспечение для маркировки данных может помочь сэкономить много времени, поэтому уделяйте ему должное внимание на ранней стадии развития проекта.

### **Остерегайтесь непрепрезентативных данных**

Модели машинного обучения способны интерпретировать только входные данные, подобные тем, что они видели раньше. Поэтому очень важно, чтобы обучающие данные были *репрезентативными* для данных, на основе которых потом модель будет вычислять прогнозы. Вы должны помнить об этом постоянно, собирая свой набор.

Представьте, что вы разрабатываете приложение, пользователь которого может фотографировать тарелку с едой, чтобы узнать название блюда. Вы обучаете модель на изображениях, полученных из популярной у гурманов социальной сети. Развортываете ее — и начинаете получать гневные отзывы пользователей: ваше приложение дает неверный ответ в восьми случаях из десяти. Но почему? Ведь точность на контрольном наборе превысила 90%! Беглый взгляд на выгруженные пользователями данные показывает, что фотографии произвольных блюд в произвольных ресторанах, сделанные с помощью случайных смартфонов, совсем не похожи на качественные снимки, на которых вы обучали модель: *ваши обучающие данные не были репрезентативными для данных, на основе которых модель вычисляет прогнозы*. Это тяжкий грех — добро пожаловать в ад машинного обучения.

### ПРОБЛЕМА СИСТЕМАТИЧЕСКОЙ ОШИБКИ ВЫБОРКИ

Особенно коварным и распространенным случаем нерепрезентативности данных является систематическая ошибка выборки. Она возникает в случаях, когда процесс сбора данных зависит от того, что вы пытаетесь предсказать, — и эта зависимость приводит к смещению измерений. Знаменитый исторический пример — случай во время президентских выборов в США 1948 года. В ночь выборов газета *Chicago Tribune* напечатала заголовок «Дьюи побеждает Трумэна», но утром были опубликованы результаты выборов, согласно которым победил Трумэн. Редактор *Tribune* поверил данным телефонного опроса, однако пользователи телефонов в 1948 году не были случайной репрезентативной выборкой населения с правом голоса. Они были более богаты, консервативны и голосовали за кандидата от республиканцев Дьюи.



«Дьюи побеждает Трумэна»: знаменитый пример систематической ошибки выборки

В настоящее время при каждом телефонном опросе учитывается возможность систематической ошибки выборки. Это не означает, что в политических опросах она осталась в прошлом — далеко не так. Но, в отличие от 1948 года, социологи знают об этом и принимают меры для ее корректировки.

Если есть возможность, собирайте данные непосредственно в том окружении, где будет использоваться модель. Модель классификации отзывов о фильмах

следует применять к новым обзорам в IMDB, а не к обзорам ресторанов Yelp или статусам в Twitter. Если вам нужно оценивать эмоциональную окраску твитов, начните со сбора и маркировки реальных твитов в той же группе пользователей, которые, как предполагается, будут использовать модель. Если нет возможности обучить модель на реальных данных, постараитесь как можно полнее определить, чем ваши обучающие данные отличаются от реальных, и устранимте эти различия.

Вы должны знать еще об одном явлении — *дрейфе понятий*. Оно встретится вам практически во всех задачах, и особенно в тех, которые основаны на данных, генерируемых пользователями. Дрейф понятий возникает, когда свойства реальных данных меняются с течением времени, приводя к постепенному снижению точности модели. Система музыкальных рекомендаций, обученная в 2013 году, в настоящее время может оказаться неэффективной. Набор данных IMDB, с которым мы работали, собран в 2011 году, и обученная на нем модель, вероятно, будет распознавать эмоциональную окраску обзоров 2020 года не так хорошо, как отзывов 2012 года, потому что словарный запас, способ выражения эмоций и жанры фильмов со временем меняются. Проблема дрейфа понятий особенно остро проявляется в контексте противоборств, таких как определение мошенничества с кредитными картами, где модели мошенничества меняются практически каждый день. В сферах с быстрым дрейфом понятий требуется постоянно собирать данные, маркировать их и повторно обучать модели.

Имейте в виду, что машинное обучение можно использовать только для запоминания шаблонов, имеющихся в обучающих данных. Распознать можно только увиденное раньше. Использование машинного обучения для предсказания будущего предполагает, что будущее будет вести себя как прошлое. Но часто это не так.

### **6.1.3. Первичный анализ данных**

Представлять набор данных как черный ящик — не лучшее решение. Прежде чем приступить к обучению моделей, обязательно проанализируйте свои данные, чтобы понять, какие особенности дают им прогнозирующую способность, что поможет обоснованно спроектировать признаки и выявить потенциальные проблемы:

- если данные включают изображения или текст на естественном языке, изучите сразу несколько образцов (и их метки);
- если данные содержат числовые признаки, постройте гистограмму распределения значений признаков, чтобы получить представление о диапазоне и частоте различных значений;
- если данные включают информацию о местоположении, нанесите ее на карту. Возможно, вы заметите какие-то явные закономерности;

- если в некоторых образцах отсутствуют значения некоторых признаков, при подготовке данных вам нужно будет решить эту проблему (подробнее о ней рассказывается в следующем разделе);
- если ваша задача связана с классификацией, подсчитайте количество образцов, представляющих каждый класс в ваших данных. Однаково ли представлены классы? Если нет, вам понадобится учесть этот дисбаланс;
- проверьте *утечку целей*: наличие в данных признаков с информацией о целях, которые могут отсутствовать в реальных данных, но потом будут использоваться для прогнозирования. Если вы обучаете на медицинских записях модель, предсказывающую вероятность заболеть раком в будущем, и записи включают признак «у этого человека диагностирован рак», то в таком случае целевые значения искусственно попадают в обучающие данные. Всегда спрашивайте себя: все ли признаки, имеющиеся в обучающих данных, будут доступны в той же форме в реальных данных?

#### **6.1.4. 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? Мера успеха будет определять все технические решения, которые вы будете принимать в процессе работы над проектом. Она должна быть прямо связана с вашими общими целями — например, такими, как успех бизнеса.

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 взять точность и полноту. Для задач ранжирования или многозначной классификации пригодится 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.

## **6.2. РАЗРАБОТКА МОДЕЛИ**

Определившись с мерой оценки прогресса, можно приступать к разработке модели. В большинстве руководств и исследовательских проектов предполагается, что разработка модели — это единственный шаг, поэтому пропускаются такие этапы, как определение задачи и сбор данных (которые, как предполагается,

уже выполнены), а также развертывание и обслуживание модели (которые, как принято считать, будут выполняться кем-то другим). На самом деле разработка модели — лишь один из множества шагов в процессе машинного обучения, и, на мой взгляд, не самый сложный. Самое сложное — это формулировка задачи, а также сбор, маркировка и очистка данных. Так что не унывайте — дальше будет проще!

## 6.2.1. Подготовка данных

Как вы уже знаете, редкие модели глубокого обучения принимают исходные данные в необработанном виде. Цель предварительной обработки — сделать исходные данные более доступными для нейронных сетей. Обработка может заключаться в векторизации, нормализации или восстановлении пропущенных значений. Многие методы предварительной обработки зависят от предметной области (например, текстовые данные и изображения обрабатываются по-разному) — мы будем рассматривать их в следующих главах в процессе обсуждения практических примеров. А пока познакомимся с основами, универсальными для всех видов данных.

### Векторизация

Желательно, чтобы все входы и цели в нейронной сети были 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 в главе 4 мы начали с того, что преобразовали текст в списки целых чисел (представляющие 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 имели векторизованную форму, поэтому мы пропустили этот шаг.

### Нормализация значений

В главе 2, в примере классификации рукописных цифр из набора MNIST, 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 предсказыванием 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 отправкой данных в сеть нам понадобилось нормализовать

каждый признак в отдельности, чтобы все они имели среднее значение, равное 0, и стандартное отклонение, равное 1.

Вообще, небезопасно передавать в нейронную сеть данные, принимающие очень большие значения (например, целые числа с большим количеством значимых разрядов, которые намного больше начальных значений, принимаемых весами сети), или разнородные данные (например, такие, 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:

- *принимать небольшие значения* — как правило, значения должны находиться в диапазоне 0–1;
- *быть однородными* — то есть все признаки должны принимать значения примерно из одного и того же диапазона.

Кроме того, может оказаться полезной (хотя и не всегда необходимой — так, мы не использовали ее в примере классификации цифр) следующая практика нормализации:

- нормализация каждого признака независимо таким образом, чтобы его среднее значение было равно 0;
- нормализация каждого признака независимо таким образом, 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. Например, в случае 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 появится недостаток значений в обучающих или контрольных данных.

От такого признака можно вообще отказаться, а можно поступить иначе.

- Если признак категориальный, то можно создать новую категорию, которая будет означать «отсутствие признака». Модель автоматически узнает, что это означает по отношению к целям.
- Если признак числовой, желательно избегать использования произвольного значения (например, 0) — это может создать разрыв в скрытом пространстве, образованном признаками, из-за чего обучаемой модели будет труднее найти

обобщающее решение. Вместо этого можно попробовать заменить отсутствующие значения средним или медианным значением для конкретного признака в наборе данных. Также можно обучить модель предсказывать отсутствующие значения одних признаков по значениям других.

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

### 6.2.2. Выбор протокола оценки

Как рассказывалось в предыдущей главе, цель модели — добиться обобщения, и каждое решение, которое вы будете принимать в процессе разработки модели, будет зависеть от *метрик на этапе проверки*, оценивающих эффективность обобщения. Цель протокола оценки — точно оценить выбранную вами меру успеха (например, точность) на реальных данных. Надежность этого процесса имеет решающее значение для построения полезной модели.

В главе 5 мы рассмотрели три распространенных протокола оценки, таких как:

- *выделение из общей выборки отдельного проверочного набора данных* — этот способ хорошо подходит при наличии большого объема данных;
- *перекрестная проверка по K блокам* — оптимальный вариант 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 возьмите один из этих вариантов. В большинстве случаев первый поможет получить достаточно надежную оценку. Однако всегда помните о *репрезентативности* проверочного набора и проявляйте осмотрительность, чтобы не допустить избыточности между обучающим и проверочным наборами.

### 6.2.3. Преодоление базового случая

Начав работу над созданием модели, ваша первая цель, как рассказывалось в главе 5, — достичь *статистической мощности*, то есть разработать небольшую модель, способную выдать более качественный результат по сравнению с базовым случаем.

На этом этапе следует сосредоточить внимание на таких трех важных аспектах, как:

- *конструирование признаков* — отфильтруйте неинформативные признаки (отбор признаков) и используйте свои знания в предметной области для конструирования новых признаков, которые могут оказаться полезными;
- *выбор правильной архитектуры* — какую архитектуру вы будете использовать: плотно связанную, сверточную, рекуррентную нейронную сеть или трансформер (Transformer)? Подходит ли в целом глубокое обучение для решения данной задачи, или лучше использовать что-то еще;
- *выбор подходящей конфигурации обучения* — какую функцию потерь, размер пакета и скорость обучения лучше использовать.

### ВЫБОР ПРАВИЛЬНОЙ ФУНКЦИИ ПОТЕРЬ

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 вычислимыми на мини-пакетах данных (в идеале 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. Следующая таблица поможет вам выбрать функцию активации для последнего уровня и функцию потерь для некоторых типичных задач.

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

Для большинства задач имеются готовые шаблоны решения — начните с них. Вы не первые, кто пытается создать детектор спама, механизм музыкальных рекомендаций или классификатор изображений. Обязательно изучите опыт

ваших предшественников. Это поможет определить методы конструирования признаков и выбрать архитектуру модели, которые с большой вероятностью подойдут для решения вашей задачи.

Обратите внимание, 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:

- гипотезу о том, что выходные данные можно предсказать по входным данным;
- гипотезу 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 проектирование с самого начала.

#### **6.2.4. Следующий шаг: 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, модель логистической регрессии будет иметь некоторую статистическую мощность для классификации цифр из набора MNIST, но этой мощности не будет достаточно, 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 пролегает эта граница, ее сначала нужно пересечь.

Для оценки того, насколько большой должна быть модель, сначала нужно сконструировать модель, обладающую эффектом переобучения. Как вы помните из главы 5, 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.

### 6.2.5. Регуляризация и настройка модели

Получив модель, обладающую статистической мощностью, и добавившись ее переобучения, вы будете спокойны в том, что движетесь в верном направлении. С этого момента вашей целью становится максимизация общности.

Этот этап занимает больше всего времени: вам придется многократно изменять свою модель, обучать ее, оценивать качество на проверочных данных (контрольные данные не должны принимать здесь никакого участия), снова изменять ее и повторять 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;
- если модель не очень большая, то 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 курирования данных или конструирования признаков: собрать больше данных и выполнить их маркировку, добавить новые признаки или удалить имеющиеся, которые не кажутся информативными.

Большую часть подобной работы можно автоматизировать с помощью *программного обеспечения для автоматической настройки гиперпараметров*, такого как KerasTuner. Эту возможность мы рассмотрим в главе 13.

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 притом, что модель напрямую не получает их). Это снижает надежность процесса оценки.

Получив удовлетворительную конфигурацию, можно обучить окончательный вариант модели на всех доступных данных (обучающих 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. Тогда можно попробовать переключиться на использование другого, более надежного протокола оценки (такого как итерационная проверка по  $K$  блокам с перемешиванием).

## 6.3. РАЗВЕРТЫВАНИЕ МОДЕЛИ

Ваша модель успешно прошла окончательную оценку на контрольном наборе и готова начать свою плодотворную деятельность.

### 6.3.1. Объяснение особенностей работы модели заинтересованным сторонам и обозначение границ ожидаемого

Успех и доверие клиентов возможны, только если модель соответствует ожиданиям или превосходит их. Фактическая система, которую вы вводите в эксплуатацию, — это только половина дела; другая половина — обозначение перед выпуском границ ожидаемого.

Неспециалисты часто имеют чересчур завышенные требования в отношении систем искусственного интеллекта. Например, они могут ожидать, что система «понимает» свою задачу и способна проявлять человеческий здравый смысл в ее контексте. Чтобы решить эту проблему, устройте демонстрацию некоторых *примеров отказа* вашей модели (например, покажите, как выглядят неправильно классифицированные образцы, особенно те, для которых неправильная классификация кажется неожиданной).

Клиенты также могут рассчитывать, что система будет работать на уровне человека, особенно если она создавалась для выполнения работы, которую раньше делали люди. Большинство моделей машинного обучения не достигают своих целей, потому что обучались на приближенных метках, созданных человеком (и потому несовершенных). Вы должны четко обозначить ожидаемые характеристики модели. Избегайте абстрактных утверждений типа «модель имеет точность 98 %» (которые большинство людей мысленно округляют до 100 %) и лучше сообщайте, например, о частоте ложноположительных и ложноотрицательных результатов. Вы могли бы сказать: «С этими настройками модель будет должно квалифицировать действия как мошеннические в 5 % случаев и пропускать фактическое мошенничество в 2,5 % случаев. Ежедневно в среднем 200 законных транзакций будут идентифицированы как мошеннические и отправлены на ручную проверку, 14 мошеннических транзакций будут пропущены и 266 будут идентифицированы верно». Четко соотнесите показатели эффективности модели с бизнес-целями.

Также обсудите с заинтересованными сторонами выбор ключевых параметров — например, порог вероятности, при котором транзакция должна отмечаться как мошенническая (разные пороги будут давать разное число ложноположительных и ложноотрицательных срабатываний). Такие решения предполагают компромиссы, которые можно учесть только при глубоком понимании бизнес-контекста.

### 6.3.2. Предоставление доступа к модели

Работа над проектом машинного обучения не заканчивается в тот момент, когда вы доберетесь до блокнота Colab и сохраните там обученную модель. Вообще, модели редко передаются в эксплуатацию в виде объекта на Python, которым вы манипулировали во время обучения.

Во-первых, вам может потребоваться экспорттировать модель в какое-то другое окружение, отличное от Python:

- если ваше промышленное окружение вообще не поддерживает Python — например, это мобильное приложение или встраиваемая система;
- если остальная часть приложения написана не на Python (а на JavaScript, C++ и т. д.) — в таком случае использование Python для обслуживания модели может повлечь значительные накладные расходы.

Во-вторых, поскольку модель будет использоваться только для прогнозирования (эта фаза называется *выводом*), а не для обучения, вы можете применить различные оптимизации, которые помогут ускорить модель и уменьшить объем используемой памяти.

Давайте кратко рассмотрим доступные варианты развертывания модели.

#### Развертывание модели в виде REST API

Это, пожалуй, самый распространенный способ предоставления доступа к модели для получения прогнозов: установите TensorFlow на сервере или в облачном экземпляре и посыпайте запросы модели через REST API. Для этого можно создать свое обслуживающее приложение, например, на основе Flask (или любой другой библиотеки для разработки веб-приложений на Python) или использовать библиотеку, входящую в состав фреймворка TensorFlow и предназначенную для предоставления доступа к моделям через API, которая называется TensorFlow Serving ([www.tensorflow.org/tfx/guide/serving](http://www.tensorflow.org/tfx/guide/serving)). TensorFlow Serving позволяет развернуть модель Keras за считанные минуты.

Данный вариант развертывания следует использовать:

- когда приложение, использующее модель для прогнозирования, имеет надежное соединение с интернетом (что очевидно, ведь приложение не сможет получать прогнозы из удаленного API, если мобильное устройство перевести в режим полета или разместить там, где доступ к интернету ограничен);
- когда приложение не имеет строгих требований к задержке: обработка запроса, вычисление прогноза и передача ответа обычно занимают около 500 миллисекунд;
- для получения прогноза не требуется передавать конфиденциальные данные, потому что данные должны быть доступны модели в расшифрованном виде (но помните, что HTTP-запросы и ответы должны шифроваться с использованием протокола SSL).

Системы поиска изображений, подбора музыкальных рекомендаций, выявления мошеннических действий с кредитными картами и анализа спутниковых изображений вполне могут обслуживать пользователей через REST API.

Перед развертыванием модели в виде REST API вам также придется ответить на очень важный вопрос: будете ли вы размещать код на своем сервере или предпочтете использовать стороннюю облачную службу. Например, Cloud AI Platform, продукт компании Google, позволяет выгрузить модель TensorFlow в Google Cloud Storage (GCS) и получить конечную точку API для отправки запросов. Платформа сама позаботится о таких тонкостях, как обслуживание пакетных прогнозов, балансировка нагрузки и масштабирование.

## Развёртывание модели на устройстве

В некоторых случаях желательно, чтобы модель работала на том же устройстве, где выполняется использующее ее приложение. Это может быть смартфон, встроенная в робота система на процессоре ARM или микроконтроллер в небольшом устройстве. Вероятно, вы видели камеру, способную автоматически обнаруживать людей и распознавать их лица: вполне возможно, что это результат работы небольшой модели глубокого обучения, действующей непосредственно в камере.

Данный вариант развертывания следует использовать, когда:

- модель имеет строгие ограничения по задержке или должна работать в отсутствие подключения к интернету. Например, в захватывающем приложении с функцией дополненной реальности задержки на ожидание ответа удаленного сервера просто недопустимы;
- модель нужно сделать достаточно маленькой, чтобы она могла работать в условиях ограниченного объема доступной памяти и на процессоре небольшой мощности. В таких случаях вам может помочь набор инструментов TensorFlow Model Optimization Toolkit ([www.tensorflow.org/model\\_optimization](http://www.tensorflow.org/model_optimization));
- точность прогнозирования не является критической для вашей задачи. Высокая точность и быстродействие — это два взаимоисключающих фактора, поэтому в условиях ограниченного объема памяти и невысокой вычислительной мощности часто приходится развертывать модель, которая не так хороша, как ее версия, требующая для работы мощный графический процессор;
- входные данные строго конфиденциальны и не должны появляться в открытом виде на удаленном сервере.

Например, модель обнаружения спама должна запускаться на смартфоне конечного пользователя в составе чат-приложения, поскольку сообщения подвергаются сквозному шифрованию и не могут быть прочитаны удаленной моделью. Точно так же модель обнаружения бракованного печенья на ленте конвейера имеет строгие ограничения по задержке и должна работать непосредственно на заводе. К счастью, в этом случае нет ограничений по мощности или объему памяти и модель можно запустить на графическом процессоре.

Для развертывания моделей Keras на смартфонах или встраиваемых устройствах можно использовать решение TensorFlow Lite ([www.tensorflow.org/lite](http://www.tensorflow.org/lite)). Этот фреймворк обеспечивает эффективную работу моделей глубокого обучения в режиме прогнозирования на смартфонах с Android и iOS, а также на компьютерах на базе ARM64, Raspberry Pi и некоторых микроконтроллерах. Он включает инструмент для преобразования моделей Keras в формат TensorFlow Lite.

### **Развертывание модели в браузере**

Модели глубокого обучения часто применяются в приложениях на JavaScript, выполняющихся в браузере или в настольной версии. Конечно, приложение можно реализовать так, что оно будет обращаться к удаленной модели через REST API, но иногда использование модели непосредственно в браузере на компьютере пользователя дает важные преимущества (с задействованием ресурсов графического процессора, если он доступен).

Данный вариант развертывания следует использовать, когда:

- вы хотите переложить нагрузку на оборудование конечного пользователя и за счет этого уменьшить нагрузку на сервер;
- входные данные должны оставаться на компьютере или телефоне конечного пользователя. Например, модель обнаружения спама в настольной и веб-версии чат-приложения (реализованного как кросс-платформенное приложение на JavaScript) должна выполняться локально;
- приложение имеет строгие ограничения по задержке. Конечно, модель, работающая на ноутбуке или смартфоне конечного пользователя, почти наверняка будет функционировать медленнее, чем на мощном графическом процессоре вашего сервера, зато ей не потребуются дополнительные 100 миллисекунд на транспортировку данных по сети;
- приложение должно продолжать работу в отсутствие подключения к интернету, после того как модель будет загружена и сохранена в локальном кэше.

Выбирайте этот вариант, только если ваша модель достаточно мала и нетребовательна к вычислительным ресурсам или объему оперативной памяти ноутбука или смартфона пользователя. Кроме того, поскольку модель будет загружена на устройство пользователя, вы должны гарантировать отсутствие в ней любой конфиденциальной информации. Помните, что из обученной модели глубокого обучения часто можно восстановить некоторую информацию об обучающих данных: если модель была обучена на конфиденциальных данных, ее лучше не выкладывать в общий доступ.

Чтобы развернуть модель на JavaScript, экосистема TensorFlow включает `TensorFlow.js` ([www.tensorflow.org/js](http://www.tensorflow.org/js)), библиотеку JavaScript для глубокого обучения, которая реализует почти все возможности Keras (изначально она разрабатывалась под рабочим названием `WebKeras`), а также множество низкоуровневых

функций TensorFlow API. Готовую модель Keras можно без особого труда импортировать в TensorFlow.js, чтобы затем использовать ее в составе браузерного приложения на JavaScript или настольного приложения Electron.

### Оптимизация обученной модели

Оптимизация обученной модели особенно важна при развертывании в окружении со строгими ограничениями на доступную вычислительную мощность и объем памяти (смартфоны и встраиваемые устройства) или с жесткими требованиями к задержке. Всегда старайтесь оптимизировать модель перед импортом в TensorFlow.js или экспортом в TensorFlow Lite.

Вот два популярных метода оптимизации, которые можно применить:

- *усечение* весов — не все коэффициенты в тензоре весов одинаково влияют на прогнозы. Порой можно значительно уменьшить количество параметров в слоях модели, сохранив только самые важные. Это поможет снизить потребление памяти и вычислительных ресурсов вашей моделью при небольшом ухудшении качества ее прогнозов. Выбирая параметры для удаления, можно контролировать соотношение размера и точности модели;
- *квантование* весов — модели глубокого обучения обучаются за счет корректировки весов с плавающей точкой одинарной точности (`float32`). Однако веса можно квантовать до 8-битных целых чисел со знаком (`int8`), чтобы получить модель исключительно для прогнозирования, которая в четыре раза меньше, но показывает точность, близкую к исходной.

Экосистема TensorFlow включает набор инструментов для усечения и квантования весов ([www.tensorflow.org/model\\_optimization](http://www.tensorflow.org/model_optimization)), глубоко интегрированный с Keras API.

### 6.3.3. Мониторинг качества работы модели в процессе эксплуатации

Итак, вы экспортировали обученную модель, интегрировали ее в свое приложение, опробовали ее на реальных данных — и она повела себя ровно так, как вы ожидали. Вы написали модульные тесты, а также реализовали журналирование и мониторинг состояния. Отлично! Теперь пришло время нажать большую красную кнопку и развернуть модель в рабочем окружении.

Но это еще не конец. После развертывания модели нужно постоянно наблюдать за ее поведением, качеством прогнозов на новых данных, взаимодействием с остальной частью приложения и ее возможным влиянием на бизнес-показатели.

- Увеличилась ли вовлеченность пользователей вашей онлайн-радиостанции после внедрения новой системы рекомендаций музыки? Увеличился ли

средний процент переходов по рекламным ссылкам после развертывания новой модели прогнозирования? Подумайте о проведении *рандомизированного A/B-тестирования*, чтобы отделить эффект влияния модели от других изменений: подмножество обращений должно обрабатываться с использованием новой модели, а другое контрольное подмножество — с применением старой процедуры. После обработки достаточно большого количества обращений разница в результатах почти наверняка будет связана с моделью.

- Если возможно, регулярно проводите ручной аудит прогнозов модели по реальным данным. Обычно при этом можно использовать ту же инфраструктуру, что и для маркировки данных: отправить некоторую часть реальных данных для маркировки вручную и сравнить прогнозы модели с новыми метками. Это обязательно следует делать, например, для системы поиска изображений и системы отбраковки печенья.
- Если аудит вручную невозможен, подумайте об альтернативных способах оценки, таких как опрос пользователей (например, в системе определения спама и оскорбительного контента).

### 6.3.4. Обслуживание модели

Наконец, ни одна модель не вечна. Вы уже знаете о *дрейфе понятий*: со временем характеристики ваших реальных данных будут меняться, постепенно снижая актуальность модели. Срок службы системы музыкальных рекомендаций будет исчисляться неделями. Системы обнаружения мошеннических действий с кредитными картами — днями. Системы поиска изображений — в лучшем случае парой лет.

После ввода модели в эксплуатацию вы должны быть готовы к обучению модели следующего поколения, которая придет на смену текущей. Для этого:

- следите, как меняются реальные данные: возможно, появятся новые признаки или потребуется расширить или иным образом изменить набор меток;
- продолжайте собирать и маркировать данные и последовательно совершенствуйте процесс маркировки. В частности, особое внимание уделяйте сбору образцов, при классификации которых текущая модель допускает много ошибок, — такие образцы, вероятнее всего, помогут повысить качество модели.

На этом мы завершаем обзор обобщенного процесса машинного обучения — он требует помнить о многом. Чтобы стать экспертом, нужны время и опыт, но не волнуйтесь: вы уже знаете намного больше, чем несколько глав назад. А теперь вы познакомились и с общей картиной — полным спектром всего, что связано с машинным обучением. Большая часть этой книги посвящена разработке моделей, но теперь вы знаете, что это лишь часть большого процесса. Всегда помните об общей картине!

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Приступая к новому проекту машинного обучения, сначала определите задачу:
  - постарайтесь понять в общем, что вы собираетесь сделать, — конечную цель и возможные ограничения;
  - соберите данные и выполните их маркировку; проанализируйте их, чтобы лучше понять суть;
  - выберите меру успеха — показатели, которые можно было бы отслеживать по проверочным данным.
- Определив задачу и собрав соответствующий набор данных, разработайте модель:
  - подготовьте данные;
  - выберите свой протокол оценки: по выделенному из общей выборки проверочному набору данных или методом перекрестной проверки по  $K$  блокам. Определите, какую часть данных вы будете использовать для проверки;
  - разработайте первую модель, более совершенную, чем базовый случай, и обладающую статистической мощностью;
  - сделайте следующий шаг: разработайте модель, способную переобучаться;
  - 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.
- Когда модель будет готова и покажет хороший результат на контрольных данных, можно приступать к ее развертыванию:
  - прежде всего объясните особенности работы модели заинтересованным сторонам и обозначьте границы ожидаемого;
  - оптимизируйте готовую модель для прогнозирования и поместите ее в выбранное окружение — на веб-сервер, мобильное устройство, в браузер, на встроенное устройство и т. д.;
  - наблюдайте за качеством работы вашей модели и продолжайте сбор данных, чтобы потом приступить к разработке модели следующего поколения.



# Работа с Keras: глубокое погружение

## В этой главе

- ✓ Создание моделей Keras с помощью класса `Sequential`, функционального API и путем наследования стандартного класса моделей.
- ✓ Использование встроенных в Keras циклов обучения и оценки.
- ✓ Использование обратных вызовов Keras для корректировки процесса обучения.
- ✓ Использование `TensorBoard` для мониторинга показателей на этапах обучения и оценки.
- ✓ Разработка цикла обучения и оценки с нуля.

У вас уже есть некоторый опыт работы с Keras — вы познакомились с моделью `Sequential`, слоями `Dense` и встроенным API обучения, оценки и прогнозирования: `compile()`, `fit()`, `evaluate()` и `predict()`. Более того, в главе 3 вы узнали, как создавать свои классы слоев, наследуя стандартный класс `Layer`, и как использовать объект `GradientTape` из библиотеки TensorFlow для реализации пошагового цикла обучения.

В следующих главах мы рассмотрим примеры задач распознавания образов, прогнозирования временных рядов, обработки естественного языка и генеративного глубокого обучения. Эти сложные приложения потребуют гораздо большего, чем могут дать последовательная архитектура `Sequential` и цикл `fit()`. Но не будем

торопиться и для начала сделаем из вас эксперта по библиотеке Keras! В этой главе вы познакомитесь с основными способами использования Keras API: со всем, что понадобится для решения более сложных задач глубокого обучения, с которыми вы столкнетесь в будущем.

## 7.1. СПЕКТР РАБОЧИХ ПРОЦЕССОВ

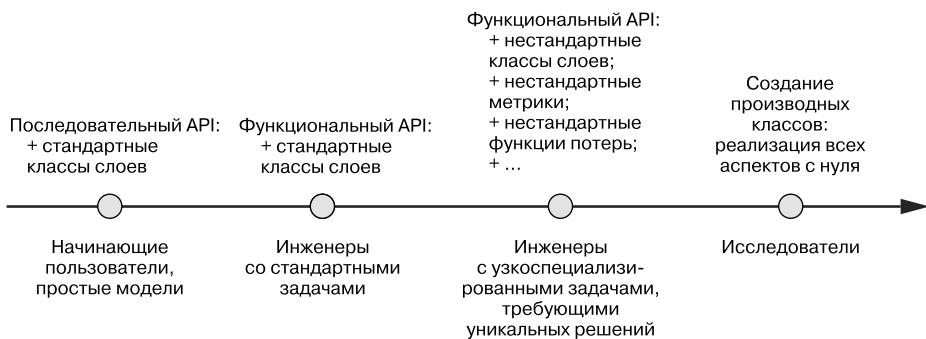
Keras API организован по принципу *постепенного раскрытия сложности*: упростить работу с библиотекой для начинающих и сохранить возможность решения задач высокой сложности, требуя лишь поэтапного обучения. Решение простых задач должно быть легким и доступным, но при этом должна сохраняться *возможность* реализации произвольно сложных рабочих процессов: какой бы узкоспециализированной ни была ваша задача, библиотека должна предоставлять четкий путь ее решения, основанный на различных приемах, которые вы освоили при изучении более простых процессов. Это означает, что новичок может вырасти до эксперта, используя одни и те же инструменты, только по-разному.

Как следствие, не существует единственно верного способа использования Keras. Вместо этого Keras предлагает целый *спектр рабочих процессов*, от очень простых до очень гибких. Keras поддерживает множество способов создания моделей и множество способов их обучения, отвечающих разным потребностям. Поскольку все эти рабочие процессы основаны на одних и тех же объектах, таких как `Layer` и `Model`, компоненты из любого рабочего процесса могут использоваться в любом другом рабочем процессе — все они способны взаимодействовать друг с другом.

## 7.2. РАЗНЫЕ СПОСОБЫ СОЗДАНИЯ МОДЕЛЕЙ KERAS

В Keras имеется три API для создания моделей (рис. 7.1):

- *последовательная модель Sequential*, наиболее доступный API — по сути, это список Python, поэтому модели данного вида ограничены простыми наборами слоев;
- *функциональный API*, ориентированный на архитектуры моделей в виде графов. Он представляет собой золотую середину в плане удобства применения и гибкости и поэтому чаще всего используется на практике;
- *наследование стандартных классов*, низкоуровневый способ, который позволяет реализовать все аспекты с нуля. Это идеальный вариант для желающих контролировать каждую мелочь. Однако при выборе данного метода у вас не будет доступа ко многим встроенным функциям Keras, а риск допустить ошибку станет выше.



**Рис. 7.1.** Принцип постепенного раскрытия сложности при создании моделей

### 7.2.1. Последовательная модель Sequential

Самый простой способ создать модель Keras — использовать уже знакомый вам класс моделей `Sequential`.

#### Листинг 7.1. Класс Sequential

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Обратите внимание, что эту модель также можно построить, последовательно вызывая метод `add()`, который действует подобно методу `append()` списков в языке Python.

#### Листинг 7.2. Последовательное создание модели Sequential

```
model = keras.Sequential()
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

В главе 4 вы узнали, что слои (точнее, их веса) создаются только в момент первого вызова. Причина подобного поведения в том, что форма слоев зависит от формы входных данных: пока форма входных данных неизвестна, слои не могут быть созданы.

В силу этого предыдущая модель `Sequential` не будет иметь весов (листинг 7.3) до передачи ей некоторых данных или до вызова ее метода `build()` с описанием формы входных данных (листинг 7.4).

**Листинг 7.3.** Непостроенные модели не имеют весов

```
>>> model.weights ← В этой точке модель еще не построена
ValueError: Weights for model sequential_1 have not yet been created.
```

**Листинг 7.4.** Первый вызов модели для ее построения

Этот вызов построит модель — после него модель будет готова принимать образцы с формой (3,). None в форме входных данных означает, что размер пакета может быть любым

```
>>> model.build(input_shape=(None, 3)) ← Теперь можно получить
>>> model.weights
[<tf.Variable "dense_2/kernel:0" shape=(3, 64) dtype=float32, ... >,
 <tf.Variable "dense_2/bias:0" shape=(64,) dtype=float32, ... >,
 <tf.Variable "dense_3/kernel:0" shape=(64, 10) dtype=float32, ... >,
 <tf.Variable "dense_3/bias:0" shape=(10,) dtype=float32, ... >]
```

После того как модель будет построена, ее содержимое можно вывести вызовом метода `summary()`, что очень удобно на этапе отладки.

**Листинг 7.5.** Метод `summary()`

```
>>> model.summary()
Model: "sequential_1"
-----  

Layer (type)           Output Shape        Param #
-----  

dense_2 (Dense)        (None, 64)          256  

-----  

dense_3 (Dense)        (None, 10)           650  

-----  

Total params: 906
Trainable params: 906
Non-trainable params: 0
```

Как видите, эта модель получила имя `sequential_1`. В Keras имя можно присвоить чему угодно — каждой модели, каждому слову.

**Листинг 7.6.** Присваивание имен моделям и слоям путем передачи аргумента `name`

```
>>> model = keras.Sequential(name="my_example_model")
>>> model.add(layers.Dense(64, activation="relu", name="my_first_layer"))
>>> model.add(layers.Dense(10, activation="softmax", name="my_last_layer"))
>>> model.build((None, 3))
>>> model.summary()
Model: "my_example_model"
```

Layer (type)	Output Shape	Param #
--------------	--------------	---------

```

my_first_layer (Dense)      (None, 64)          256
=====
my_last_layer (Dense)       (None, 10)           650
=====
Total params: 906
Trainable params: 906
Non-trainable params: 0
=====
```

При пошаговом построении модели `Sequential` удобно иметь возможность посмотреть на ее текущее состояние после добавления очередного слоя. Но сводку невозможно получить, пока модель не построена! Этую проблему можно решить, строя модель `Sequential` на лету, для чего достаточно заранее объявить форму входных данных. Это можно сделать с помощью класса `Input`.

**Листинг 7.7.** Предварительное определение формы входных данных модели с помощью класса `Input`

```

model = keras.Sequential()
model.add(keras.Input(shape=(3,)))
model.add(layers.Dense(64, activation="relu"))
```

Использование `Input` для объявления формы входных данных. Обратите внимание, что аргумент `shape` должен определять форму одного образца, но не пакета

Теперь вы сможете вызывать `summary()` и наблюдать, как меняется форма выходных данных модели по мере добавления дополнительных слоев:

```
>>> model.summary()
Model: "sequential_2"
```

```

Layer (type)          Output Shape         Param #
=====
dense_4 (Dense)      (None, 64)           256
=====
Total params: 256
Trainable params: 256
Non-trainable params: 0
=====
```

```
>>> model.add(layers.Dense(10, activation="softmax"))
>>> model.summary()
Model: "sequential_2"
```

```

Layer (type)          Output Shape         Param #
=====
dense_4 (Dense)      (None, 64)           256
=====
dense_5 (Dense)      (None, 10)            650
=====
Total params: 906
Trainable params: 906
Non-trainable params: 0
=====
```

Это довольно распространенный прием отладки при работе со слоями, которые применяют сложные преобразования к своим входным данным (например, со сверточными слоями, о которых рассказывается в главе 8).

## 7.2.2. Функциональный API

Модель `Sequential` проста в использовании, но круг сфер ее применения чрезвычайно ограничен: она может выражать модели только с одним входом и одним выходом, последовательно применяя слои друг за другом. Но на практике довольно часто встречаются модели с несколькими входами (например, изображение и его метаданные), несколькими выходами (разные признаки, которые необходимо предсказать) или нелинейной топологией.

В таких случаях модель следует строить с помощью функционального API — именно такой подход для большинства моделей Keras вы встретите в действительности. Это мощный и увлекательный способ конструирования, напоминающий игру с кубиками лего.

### Простой пример

Начнем с чего-нибудь простого — например, воспроизведем последовательность из двух слоев, что была создана в предыдущем разделе. Следующий листинг иллюстрирует создание этой модели с помощью функционального API.

#### Листинг 7.8. Создание простой модели с двумя слоями `Dense` с помощью функционального API

```
inputs = keras.Input(shape=(3,), name="my_input")
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Рассмотрим этот процесс шаг за шагом.

Сначала объявляется форма входных данных в виде экземпляра класса `Input` ( обратите внимание, что этим объектам, как и любым другим, тоже можно присваивать имена):

```
inputs = keras.Input(shape=(3,), name="my_input")
```

Объект `input` хранит информацию о форме и типе данных, которые будет обрабатывать модель:

```
>>> inputs.shape
(None, 3)
>>> inputs.dtype
float32
```

Модель будет обрабатывать пакеты, в которых каждый образец имеет форму (3,). Количество образцов в пакете может меняться (о чем говорит значение `None`, определяющее размер пакета)

Данные в пакетах имеют тип `float32`

Такие объекты называются *символическими тензорами*. Они не содержат никаких действительных данных, но определяют параметры фактических тензоров, которые модель будет получать на входе. То есть они *представляют* будущие тензоры данных.

Затем создается слой и при создании ему передается информация о входных данных:

```
features = layers.Dense(64, activation="relu")(inputs)
```

Любым слоям Keras могут передаваться тензоры с реальными данными или такие вот символические тензоры. В последнем случае они возвращают новый символический тензор с информацией о форме и типе выходных данных:

```
>>> features.shape  
(None, 64)
```

После получения информации о выходных данных последнего слоя создается экземпляр модели; при этом конструктору модели передаются сведения о входах и выходах.

```
outputs = layers.Dense(10, activation="softmax")(features)  
model = keras.Model(inputs=inputs, outputs=outputs)
```

Вот сводная информация о получившейся модели:

```
>>> model.summary()  
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
<hr/>		
my_input (InputLayer)	[(None, 3)]	0
dense_6 (Dense)	(None, 64)	256
dense_7 (Dense)	(None, 10)	650
<hr/>		
Total params: 906		
Trainable params: 906		
Non-trainable params: 0		

---

## Модели с несколькими входами и выходами

В отличие от этой простой модели большинство моделей глубокого обучения не похожи на списки и скорее напоминают графы. Например, они могут иметь несколько входов или несколько выходов. Именно благодаря таким моделям функциональный API предстает во всем блеске.

Представьте, что вы создаете систему для ранжирования заявок в службу поддержки клиентов по важности и для распределения между соответствующими отделами. Ваша модель имеет три входа:

- название заявки (текстовый вход);
- текст с описанием заявки (текстовый вход);
- теги, добавленные пользователем (категориальный вход; предполагается, что теги представлены в формате прямого кодирования).

Входные текстовые данные можно закодировать в виде массивов нулей и единиц с размерами `vocabulary_size` (мы рассмотрим методы кодирования текста подробнее в главе 11).

Модель также имеет два выхода:

- оценку важности заявки — скалярное значение от 0 до 1 (сигмоидный выход);
- отдел, который должен обработать заявку (результат применения функции `softmax` к множеству отделов).

С помощью функционального API данную модель можно построить несколькими строками кода.

#### Листинг 7.9. Создание модели с несколькими входами и выходами с помощью функционального API

```
vocabulary_size = 10000
num_tags = 100
num_departments = 4

title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape=(vocabulary_size,), name="text_body")
tags = keras.Input(shape=(num_tags,), name="tags")

features = layers.concatenate([title, text_body, tags])
features = layers.Dense(64, activation="relu")(features)

priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department")(features)

model = keras.Model(inputs=[title, text_body, tags],
                     outputs=[priority, department])
```

**Объединение входных признаков в один тензор features**

**Определение входов модели**

**Добавление промежуточного слоя для рекомбинации входных признаков в более богатые представления**

**Определение выходов модели**

**Создание модели с передачей ей информации о входах и выходах**

Функциональный API — это простой, как конструктор лего, но очень гибкий способ определения произвольных графов слоев, подобных этому.

## Обучение модели с несколькими входами и выходами

Обучаются модели с несколькими входами и выходами почти так же, как модели `Sequential`, — вызовом функции `fit()` со списками входных и выходных данных. Эти списки должны передаваться в том же порядке, в каком информация о входных данных передавалась конструктору модели.

### Листинг 7.10. Обучение модели с передачей массивов входных данных и целей

```
import numpy as np
num_samples = 1280
title_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
text_body_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
tags_data = np.random.randint(0, 2, size=(num_samples, num_tags))

priority_data = np.random.random(size=(num_samples, 1))
department_data = np.random.randint(0, 2, size=(num_samples, num_departments))

model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[[["mean_absolute_error"], ["accuracy"]]])
model.fit([title_data, text_body_data, tags_data],
          [priority_data, department_data],
          epochs=1)
model.evaluate([title_data, text_body_data, tags_data],
               [priority_data, department_data])
priority_preds, department_preds = model.predict(
    [title_data, text_body_data, tags_data])
```

Чтобы не зависеть от конкретного порядка передачи аргументов (например, у вашей модели много входов или выходов и вам не хотелось бы в них запутаться), можно также использовать имена, присвоенные объектам `Input` и выходным слоям, и передавать данные через словарь.

### Листинг 7.11. Обучение модели с передачей массивов входных данных и целей в словаре

```
model.compile(optimizer="rmsprop",
              loss={"priority": "mean_squared_error",
                    "department": "categorical_crossentropy"},
```

```

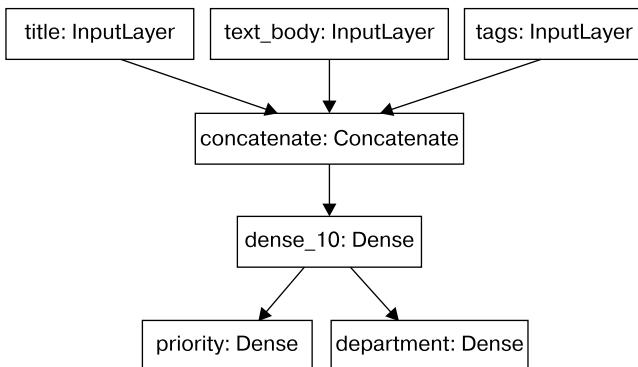
metrics={"priority": ["mean_absolute_error"],
         "department": ["accuracy"]})
model.fit({"title": title_data, "text_body": text_body_data,
           "tags": tags_data},
           {"priority": priority_data, "department": department_data},
           epochs=1)
model.evaluate({"title": title_data, "text_body": text_body_data,
                "tags": tags_data},
                {"priority": priority_data, "department": department_data})
priority_preds, department_preds = model.predict(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})

```

## Мощь функционального API: доступ к информации о связях между слоями

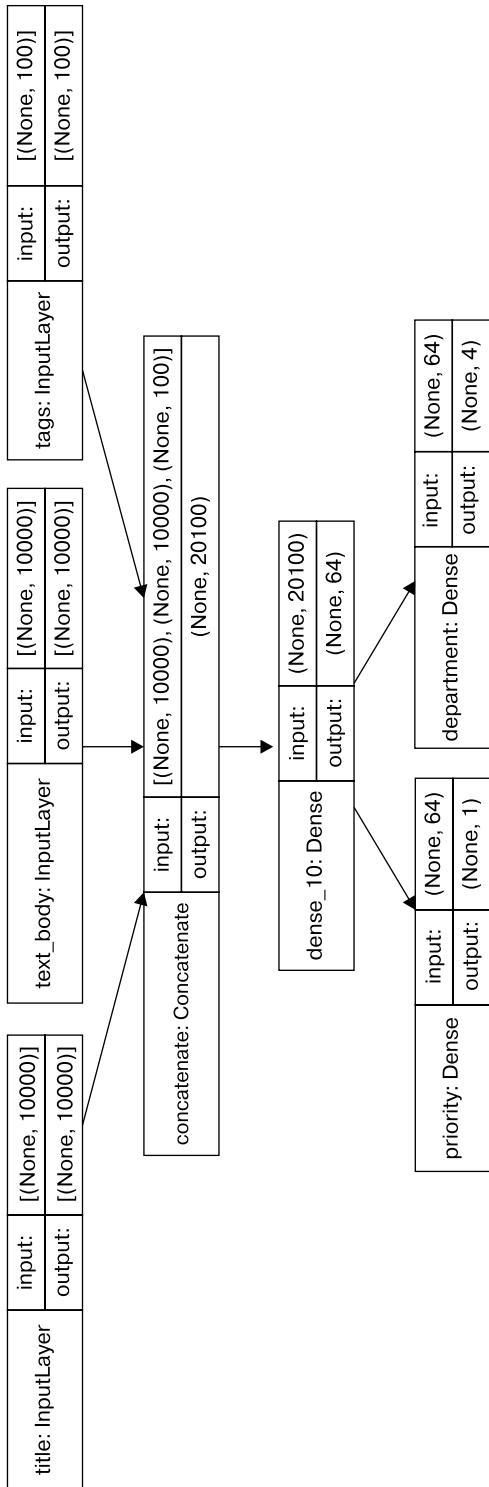
Функциональная модель — это графовая структура данных. Она позволяет исследовать связи между слоями и повторно использовать предыдущие узлы графа (которые являются выходами слоев) в новых моделях. Кроме того, она хорошо соответствует «ментальной модели» (графу слоев), которую используют многие исследователи, рассуждая о глубоких нейронных сетях, а также поддерживает два важных способа использования: визуализацию модели и извлечение признаков.

Давайте отобразим связи в только что созданной модели (ее *топологию*). Получить изображение функциональной модели в виде графа можно с помощью утилиты `plot_model()` (рис. 7.2).



**Рис. 7.2.** Графическое представление модели классификации заявок, сгенерированное функцией `plot_model()`

В эту диаграмму можно добавить форму входа и выхода каждого слоя в модели, что может пригодиться в процессе отладки (рис. 7.3).



**Рис. 7.3.** Графическое представление модели с информацией о форме входов и выходов

Слово `None` в формах тензоров представляет размер пакета: данная модель может обрабатывать пакеты любого размера.

Доступность информации о связях между слоями также означает возможность исследовать и повторно использовать отдельные узлы (выходы слоев) в графе. Свойство модели `model.layers` возвращает список слоев в модели, и для каждого слоя в этом списке можно исследовать их свойства `layer.input` и `layer.output`.

### Листинг 7.12. Получение входов и выходов слоя в функциональной модели

```
>>> model.layers
[<tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d358>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d2e8>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d470>,
 <tensorflow.python.keras.layers.merge.Concatenate at 0x7fa963f9d860>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa964074390>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f9d898>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f95470>]
>>> model.layers[3].input
[<tf.Tensor "title:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "text_body:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "tags:0" shape=(None, 100) dtype=float32>]
>>> model.layers[3].output
<tf.Tensor "concatenate(concat:0" shape=(None, 20100) dtype=float32>
```

Это позволяет производить *извлечение признаков* в моделях, повторно использующих промежуточные признаки из другой модели.

Представьте, что вы решили добавить еще один выход в предыдущую модель — оценку времени, которое потребуется для решения обозначенной в заявке проблемы (своего рода рейтинг сложности). Это можно сделать с помощью слоя классификации по трем категориям: «простая» проблема, «средняя» и «сложная». Причем вам не нужно воссоздавать и повторно обучать модель с нуля: можно начать с промежуточных признаков предыдущей модели, как показано ниже.

### Листинг 7.13. Создание новой модели, повторно использующей выходы промежуточных слоев

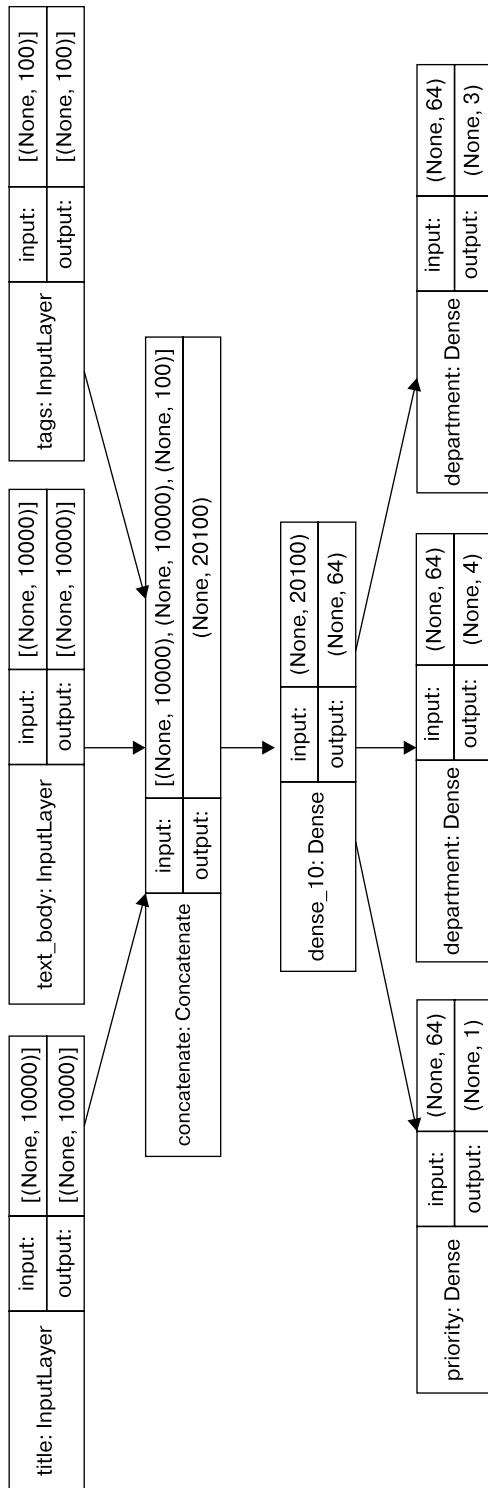
```
features = model.layers[4].output
difficulty = layers.Dense(3, activation="softmax", name="difficulty")(features)

new_model = keras.Model(
    inputs=[title, text_body, tags],
    outputs=[priority, department, difficulty])
```

↑  
layers[4] — это промежуточный  
слой Dense

Теперь получим графическое представление новой модели (рис. 7.4):

```
keras.utils.plot_model(
    new_model, "updated_ticket_classifier.png", show_shapes=True)
```



**Рис. 7.4.** Графическое представление новой модели

### 7.2.3. Создание производных от класса Model

Последний и наиболее продвинутый подход к созданию моделей, о котором вы должны знать, — создание производных от класса `Model`. В главе 3 рассказывалось, как создать подкласс класса `Layer`, чтобы получить свой класс слоев. Подклассы класса `Model` создаются похожим образом:

- в методе `__init__()` определяются слои, которые будет использовать модель;
- в методе `call()` с помощью созданных ранее слоев определяется порядок выполнения прямого прохода модели;
- создается экземпляр вашего подкласса, после чего ему передаются данные для создания весов.

#### Реализация предыдущего примера созданием производного класса от класса Model

Рассмотрим простой пример: реализуем модель управления заявками в службу поддержки клиентов, создав производный класс от класса `Model`.

##### Листинг 7.14. Простой подкласс моделей

```
class CustomerTicketModel(keras.Model):
    def __init__(self, num_departments):
        super().__init__()
        self.concat_layer = layers.concatenate()
        self.mixing_layer = layers.Dense(64, activation="relu")
        self.priority_scoring = layers.Dense(1, activation="sigmoid")
        self.department_classifier = layers.Dense(
            num_departments, activation="softmax")

    def call(self, inputs):
        title = inputs["title"]
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)

        priority = self.priority_scoring(features)
        department = self.department_classifier(features)
        return priority, department
```

Определив свой класс моделей, можно создать его экземпляр. Обратите внимание, что веса будут созданы только при первом вызове модели с некоторыми данными, так же как в подклассах `Layer`:

```
model = CustomerTicketModel(num_departments=4)

priority, department = model(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})
```

Пока не видно никаких отличий от подклассов `Layer`, которые мы создавали в главе 3. В чем же тогда разница между подклассом `Layer` и подклассом `Model`? Все просто: слой — это просто блок, используемый при строительстве моделей, а модель — объект более высокого уровня, который вы будете обучать, экспортить для прогнозирования и т. д. Проще говоря, класс `Model` имеет методы `fit()`, `evaluate()` и `predict()`, а у слоев этих методов нет. В остальном данные два класса практически идентичны. (Еще одно отличие: модель можно *сохранить* в файл на диске, о чём мы расскажем несколькими разделами позже.)

Компиляция и обучение подкласса `Model` производятся точно так же, как компиляция последовательных или функциональных моделей:

```
Структура аргументов, передаваемых в параметрах loss
и metrics, должна точно соответствовать тому, что возвращает
call() — в данном случае это списки с двумя элементами

model.compile(optimizer="rmsprop",
               loss=["mean_squared_error", "categorical_crossentropy"],
               metrics=[["mean_absolute_error"], ["accuracy"]])
model.fit({"title": title_data,
           "text_body": text_body_data,
           "tags": tags_data},
           [priority_data, department_data],
           epochs=1)
model.evaluate({"title": title_data,
                "text_body": text_body_data,
                "tags": tags_data},
               [priority_data, department_data])
priority_preds, department_preds = model.predict({"title": title_data,
                                                   "text_body": text_body_data,
                                                   "tags": tags_data})
```

Структура входных данных должна точно соответствовать структуре параметров метода `call()` — в данном случае это словарь с ключами `title`, `text_body` и `tags`

Структура цели должна точно соответствовать тому, что возвращает метод `call()`, — в данном случае это список с двумя элементами

Прием, основанный на создании подклассов класса `Model` — наиболее гибкий способ построения модели. Он позволяет конструировать модели, которые нельзя выразить в форме ориентированного ациклического графа слоев; представьте, например, модель, в которой метод `call()` использует слои внутри цикла `for` или даже вызывает их рекурсивно. В подклассе `Model` такое возможно — главные здесь вы.

### **Будьте внимательны: что не поддерживают подклассы класса Model**

За такую свободу приходится платить: определяя свои подклассы класса `Model`, вы отвечаете за реализацию большей части логики работы модели, поэтому вероятность допустить ошибку намного больше. Как следствие, вам придется больше времени потратить на отладку. Теперь вы разрабатываете новый объект на Python, а не просто собираете вместе кубики лего.

Кроме того, модели на основе подклассов по своей природе существенно отличаются от функциональных моделей. Функциональная модель — это явная структура данных: граф слоев, который можно просматривать, исследовать и изменять. Модель на основе подкласса — это фрагмент байт-кода, класс Python с методом `call()`, который содержит нестандартный код. Подклассы дают дополнительную гибкость — вы можете реализовать любые функции, какие только пожелаете. Но также это накладывает определенные ограничения.

Например, поскольку способ соединения слоев друг с другом скрыт в теле метода `call()`, вы не сможете получить доступ к этой информации. Вызов `summary()` не покажет связи между слоями, а `plot_model()` не сможет построить топологию модели. Точно так же вы не доберетесь к узлам графа слоев в модели на основе подкласса, чтобы извлечь признаки, потому что графа просто не существует. После создания экземпляра модели логика ее прямого прохода становится настоящим черным ящиком.

#### 7.2.4. Смешивание и согласование различных компонентов

Важно отметить, что выбор одного из описанных подходов к созданию моделей — с помощью класса `Sequential`, функционального API или путем создания подкласса класса `Model` — не препятствует использованию других подходов. Все модели в Keras API способны беспрепятственно взаимодействовать друг с другом, будь то модели `Sequential`, функциональные модели или подклассы класса `Model`, написанные с нуля. Все они являются частью общего спектра рабочих процессов.

Например, подкласс класса `Layer` или `Model` можно использовать в функциональной модели.

**Листинг 7.15.** Создание функциональной модели, включающей подкласс класса `Model`

```
class Classifier(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        if num_classes == 2:
            num_units = 1
            activation = "sigmoid"
        else:
            num_units = num_classes
            activation = "softmax"
        self.dense = layers.Dense(num_units, activation=activation)

    def call(self, inputs):
        return self.dense(inputs)
```

```
inputs = keras.Input(shape=(3,))
features = layers.Dense(64, activation="relu")(inputs)
outputs = Classifier(num_classes=10)(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```

И наоборот, функциональную модель можно применить в подклассе класса `Layer` или `Model`.

**Листинг 7.16.** Создание подкласса класса `Model`, использующего функциональную модель

```
inputs = keras.Input(shape=(64,))
outputs = layers.Dense(1, activation="sigmoid")(inputs)
binary_classifier = keras.Model(inputs=inputs, outputs=outputs)

class MyModel(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        self.dense = layers.Dense(64, activation="relu")
        self.classifier = binary_classifier

    def call(self, inputs):
        features = self.dense(inputs)
        return self.classifier(features)

model = MyModel()
```

## 7.2.5. Используйте правильный инструмент

Вы познакомились со спектром рабочих процессов построения моделей Keras: от простейшего процесса создания последовательной модели `Sequential` до самого продвинутого, основанного на создании подклассов класса `Model`. Но в каких случаях лучше использовать тот или другой подход? У каждого из них есть свои плюсы и минусы – выбирайте, исходя из конкретной стоящей перед вами задачи.

Обычно функциональный API является хорошим компромиссным решением в плане простоты использования и гибкости. Он открывает прямой доступ к связям между слоями, что может пригодиться при отладке процедуры построения модели или для извлечения признаков. Если есть возможность взять функциональный API – например, если модель можно выразить в виде ориентированного ациклического графа слоев, – рекомендую ею воспользоваться вместо создания подклассов класса `Model`.

В дальнейшем все примеры в этой книге будут использовать функциональный API просто потому, что все модели, с которыми мы будем работать, могут

быть представлены в виде графов слоев. Однако мы будем часто использовать подклассы слоев. В целом функциональные модели, включающие подклассы, позволяют сочетать лучшее из обоих миров: высокую гибкость разработки при сохранении преимуществ функционального API.

## 7.3. ВСТРОЕННЫЕ ЦИКЛЫ ОБУЧЕНИЯ И ОЦЕНКИ

Принцип постепенного раскрытия сложности — доступ к спектру рабочих процессов шаг за шагом, от предельно простых до бесконечно гибких — также применим для обучения моделей. Библиотека Keras предлагает разные подходы к обучению моделей, от несложных, таких как вызов `fit()` с обучающими данными, до продвинутых, связанных с разработкой новых алгоритмов обучения с нуля.

Вы уже знакомы с последовательностью вызовов `compile()`, `fit()`, `evaluate()`, `predict()`. Чтобы вспомнить ее, взгляните на следующий листинг.

**Листинг 7.17.** Стандартный рабочий процесс: `compile()`, `fit()`, `evaluate()`, `predict()`

```
from tensorflow.keras.datasets import mnist
def get_mnist_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)

    model = keras.Model(inputs, outputs)
    return model

(images, labels), (test_images, test_labels) = mnist.load_data()
images = images.reshape((60000, 28 * 28)).astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28)).astype("float32") / 255
train_images, val_images = images[10000:], images[:10000]
train_labels, val_labels = labels[10000:], labels[:10000]

model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=3,
          validation_data=(val_images, val_labels))
test_metrics = model.evaluate(test_images, test_labels)
predictions = model.predict(test_images)
```

Creation of the model (moved to a separate function to have the possibility to use it later)

Data loading for training and testing

Compilation of the model with the specified optimizer, loss function, and metrics

Call to fit() for training the model using training data, specifying validation data

Call to evaluate() to calculate losses and metrics on control samples

Call to predict() with control samples for calculating probabilities of belonging to a particular class

Call to fit() for training the model; it is possible to pass validation data for monitoring quality of the model on data different from training ones

Call to predict() with control samples for calculating probabilities of belonging to a particular class

Данный простой процесс можно скорректировать:

- указав свои метрики;
- передав *функции обратного вызова* методу `fit()`, чтобы выполнить некоторые действия в определенные моменты обучения.

Посмотрим, как это сделать.

### 7.3.1. Использование собственных метрик

Метрики являются ключом к оценке качества модели — в частности, они позволяют измерить разницу качества модели на обучающих и контрольных данных. Метрики, обычно используемые для классификации и регрессии, уже включены в стандартный модуль `keras.metrics`, и в большинстве случаев вы будете брать именно их. Но иногда, особенно при решении необычных задач, вам может понадобиться умение писать свои метрики. Это просто!

Метрики в Keras являются подклассом класса `keras.metrics.Metric`. Подобно слоям, метрики имеют внутреннее состояние, хранящееся в переменных TensorFlow. Но, в отличие от слоев, эти переменные не обновляются на этапе обратного распространения, поэтому вам придется написать свою логику их обновления в методе `update_state()`.

Вот пример простой метрики, измеряющей среднеквадратичную ошибку (Root Mean Squared Error, RMSE).

**Листинг 7.18.** Реализация метрики путем создания класса, производного от класса Metric

```
import tensorflow as tf
class RootMeanSquaredError(keras.metrics.Metric): ←
    Подкласс
    класса Metric
    def __init__(self, name="rmse", **kwargs):
        super().__init__(name=name, **kwargs)
        self.mse_sum = self.add_weight(name="mse_sum", initializer="zeros")
        self.total_samples = self.add_weight(
            name="total_samples", initializer="zeros", dtype="int32")
```

```
def update_state(self, y_true, y_pred, sample_weight=None): ←
    y_true = tf.one_hot(y_true, depth=tf.shape(y_pred)[1]) ←
    mse = tf.reduce_sum(tf.square(y_true - y_pred))
    self.mse_sum.assign_add(mse)
    num_samples = tf.shape(y_pred)[0]
    self.total_samples.assign_add(num_samples)
```

Согласно нашей модели  
MNIST ожидаются  
категориальные прогнозы  
и целочисленные метки

Определение в конструкторе переменных  
для хранения состояния. По аналогии  
со слоями есть возможность использовать  
метод `add_weight()`

`update_state()` реализует логику обновления состояния. Аргумент  
`y_true` — это цели (или метки) для одного пакета, а `y_pred`  
представляет соответствующие прогнозы модели. Аргумент  
`sample_weight` можно игнорировать — здесь он не используется

Для получения текущего значения метрики нужно реализовать метод `result()`:

```
def result(self):
    return tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))
```

Также следует предоставить возможность сбросить метрику в исходное состояние без необходимости создавать ее повторно. Это позволит использовать одни и те же объекты метрик в разные эпохи обучения или на этапах и обучения, и оценки. Для этого достаточно реализовать метод `reset_state()`:

```
def reset_state(self):
    self.mse_sum.assign(0.)
    self.total_samples.assign(0)
```

Нестандартные метрики используются точно так же, как встроенные. Давайте протестируем нашу метрику:

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy", RootMeanSquaredError()])
model.fit(train_images, train_labels,
          epochs=3,
          validation_data=(val_images, val_labels))
test_metrics = model.evaluate(test_images, test_labels)
```

Теперь индикатор выполнения `fit()` будет отображать среднеквадратичную ошибку модели.

### 7.3.2. Использование обратных вызовов

Запуск процедуры обучения продолжительностью в десятки эпох на большом наборе данных вызовом `model.fit()` 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 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. Поддержка *обратных вызовов* в Keras поможет превратить вызов `model.fit()` из бумажного самолетика в интеллектуальный, автономный беспилотник, способный оценивать свое состояние и своевременно выполнять управляющие действия.

*Обратный вызов* — это объект (экземпляр класса, реализующего конкретные методы), который передается в модель через вызов `fit()` и который будет вызываться моделью в разные моменты обучения. Он имеет доступ ко всей информации о состоянии модели и ее качестве и может предпринимать следующие

действия: прерывать обучение, сохранять модель, загружать разные наборы весов или как-то иначе изменять состояние модели.

Вот несколько примеров использования обратных вызовов:

- *фиксация состояния модели в контрольных точках* — сохранение текущего состояния 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 `fit()`, с которым вы уже знакомы, — это на самом деле обратный вызов!

Модуль `keras.callbacks` включает 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 понять, как ими пользоваться: `EarlyStopping` и `ModelCheckpoint`.

### Обратные вызовы `EarlyStopping` и `ModelCheckpoint`

Многие аспекты обучения модели нельзя предсказать заранее — например, количество эпох, обеспечивающее оптимальное значение потерь на проверочном наборе. В примерах, приводившихся до сих пор, использовалась стратегия обучения с достаточно большим количеством эпох. Таким способом достигался эффект переобучения: когда сначала выполнялся первый прогон, чтобы выяснить необходимое количество эпох обучения, а затем второй — новый — с этим количеством. Конечно, данная стратегия довольно расточительная. Гораздо лучше было бы остановить обучение, как только выяснится, что оценка потерь на проверочном наборе перестала улучшаться. Это можно реализовать с использованием обратного вызова `EarlyStopping`.

Обратный вызов `EarlyStopping` прерывает процесс обучения, если находящаяся под наблюдением целевая метрика не улучшалась на протяжении заданного количества эпох. Он позволит остановить обучение после наступления эффекта

переобучения и тем самым избежать повторного обучения модели для меньшего количества эпох. Данный обратный вызов обычно используется в комбинации с обратным вызовом `ModelCheckpoint`, который может сохранять состояние модели в ходе обучения (и при необходимости сохранять только лучшую модель: версию, достигшую лучшего качества к концу эпохи):

#### Листинг 7.19. Использование параметра callbacks метода fit()

```
Обратные вызовы передаются в модель
через параметр callbacks метода fit()
в виде списка. Вы можете передать любое
количество обратных вызовов
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path.keras",
        monitor="val_loss",
        save_best_only=True,
    )
]
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=callbacks_list,
          validation_data=(val_images, val_labels))

Сохраняет
текущие веса
после каждой
эпохи
Путь
к файлу
модели
Прерывает обучение,
когда качество
модели перестает
улучшаться
Следит за изменением
точности модели
на проверочных данных
Прерывает обучение,
если точность
не улучшается
в течение двух эпох
Эти два аргумента требуют, чтобы файл
модели не перезаписывался, если значение
val_loss не улучшилось, что позволяет
сохранить только лучшую модель
Мы следим за точностью,
поэтому она должна быть частью
набора метрик модели

Обратите внимание: поскольку обратный вызов следует
за потерями и точностью на проверочных данных,
мы должны передать validation_data в вызов fit()
```

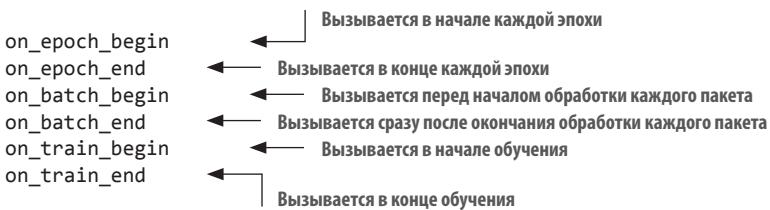
Помните, что модель всегда можно сохранить вручную после обучения: нужно лишь вызвать метод `model.save('путь_к_файлу')`. Чтобы загрузить сохраненную модель, просто примените:

```
model = keras.models.load_model("checkpoint_path.keras")
```

#### 7.3.3. 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 создания подкласса класса `keras.callbacks.Callback`. Вы можете реализовать любые из следующих

методов с говорящими именами, которые будут вызываться в соответствующие моменты в ходе 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.). Методам `on_epoch_*` и `on_batch_*` также передается индекс эпохи или пакета в первом аргументе (целое число).

Вот простой пример обратного вызова, который сохраняет список значений потерь для каждого пакета во время обучения и график изменения потерь в конце каждой эпохи.

#### Листинг 7.20. Создание своего обратного вызова наследованием класса Callback

```
from matplotlib import pyplot as plt

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs):
        self.per_batch_losses = []

    def on_batch_end(self, batch, logs):
        self.per_batch_losses.append(logs.get("loss"))

    def on_epoch_end(self, epoch, logs):
        plt.clf()
        plt.plot(range(len(self.per_batch_losses)), self.per_batch_losses,
                 label="Потери на обучающих данных для каждого пакета")
        plt.xlabel(f"Пакеты (эпоха {epoch})")
        plt.ylabel("Потери")
        plt.legend()
        plt.savefig(f"plot_at_epoch_{epoch}")
        self.per_batch_losses = []
```

Испытаем его:

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=[LossHistory()],
          validation_data=(val_images, val_labels))
```

Сохраненный график можно увидеть на рис. 7.5.

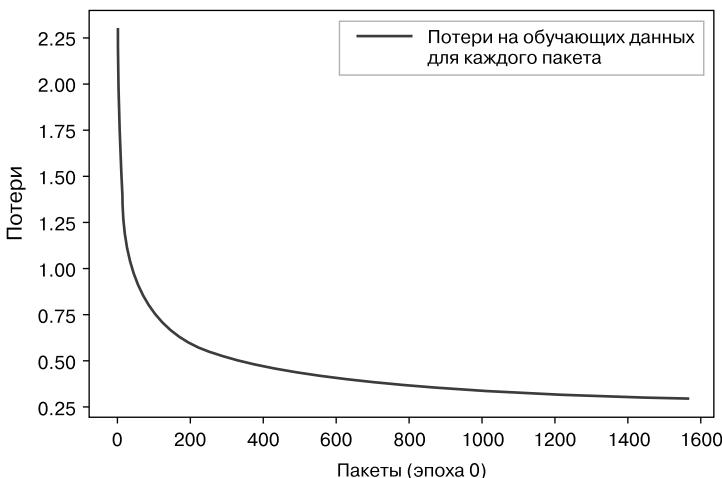


Рис. 7.5. График, созданный нашим собственным обратным вызовом

### 7.3.4. Мониторинг и визуализация с помощью TensorBoard

Для проведения результативных исследований или разработки качественных моделей необходимо иметь разностороннюю, часто обновляющуюся информацию о происходящем внутри модели в ходе экспериментов. В этом суть экспериментов — получить информацию (как можно больше информации) о том, насколько хорошо работает модель. Движение вперед носит итеративный, или циклический, характер. Вы начинаете с идеи и разрабатываете план эксперимента, который подтвердит или опровергнет ее. Далее вы запускаете эксперимент и обрабатываете полученную информацию. Это дает толчок к рождению новой идеи. И чем больше итераций в данном цикле вы выполните, тем совершеннее и мощнее будут становиться ваши идеи. Keras поможет вам перейти от идеи к эксперименту в кратчайшие сроки, а с помощью GPU вы получите результаты эксперимента достаточно быстро. Но как быть с обработкой результатов? Здесь вам пригодится TensorBoard (рис. 7.6).

TensorBoard ([www.tensorflow.org/tensorboard](http://www.tensorflow.org/tensorboard)) — браузерное приложение, которое можно запускать локально. Это лучший способ наблюдения за происходящим внутри модели во время обучения. TensorBoard позволяет:

- визуально контролировать метрики в процессе обучения;
- отображать архитектуру модели;
- выводить гистограммы активаций и градиентов;
- исследовать векторные представления в трехмерной системе координат.



**Рис. 7.6.** Циклическое движение вперед

Самый простой способ использовать TensorBoard с моделью Keras и методом `fit()` – определить обратный вызов `keras.callbacks.TensorBoard`.

В простейшем случае достаточно указать, куда должна записываться информация этим обратным вызовом, и все:

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
tensorboard = keras.callbacks.TensorBoard(
    log_dir="/full_path_to_your_log_dir",
)
model.fit(train_images, train_labels,
          epochs=10,
          validation_data=(val_images, val_labels),
          callbacks=[tensorboard])
```

С началом обучения модель будет записывать информацию в указанное место положение. Если обучение выполняется на локальном компьютере, то вы можете запустить локальный сервер TensorBoard следующей командой (обратите внимание, что выполняемый файл `tenorboard` уже должен быть доступен, если библиотека TensorFlow устанавливалась с помощью pip; если нет, можно установить TensorBoard вручную командой `pip install tensorboard`):

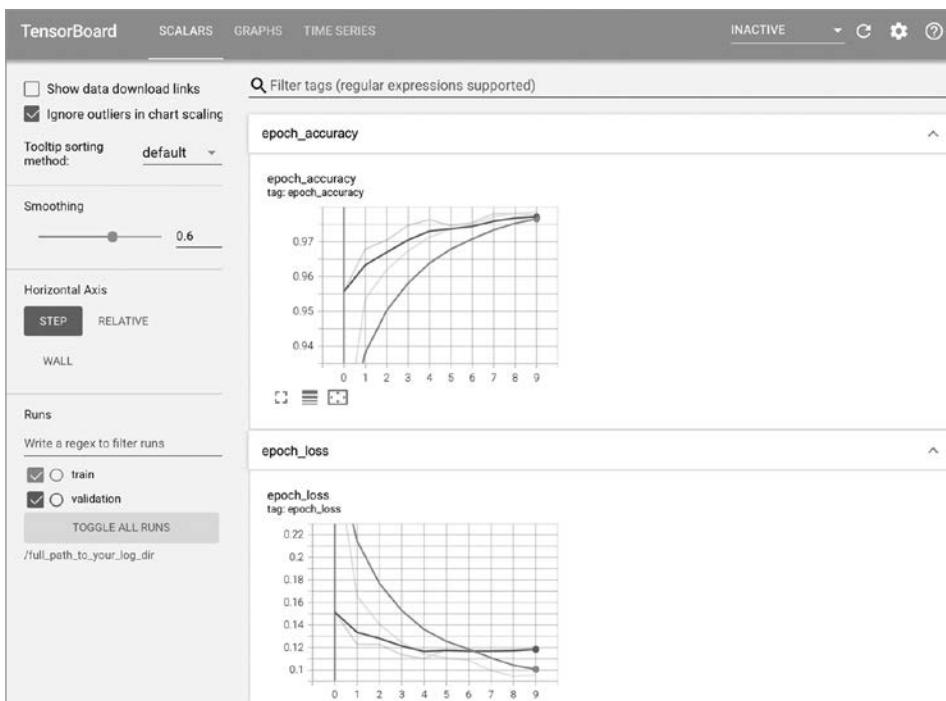
```
tensorboard --logdir /full_path_to_your_log_dir
```

Данная команда выведет URL, который затем можно ввести в адресную строку браузера, чтобы получить доступ к интерфейсу TensorBoard.

Если обучение производится в блокноте Colab, то можно запустить встроенный экземпляр TensorBoard в блокноте, выполнив следующую команду:

```
%load_ext tensorboard
%tensorboard --logdir /full_path_to_your_log_dir
```

В интерфейсе TensorBoard можно наблюдать в режиме реального времени, как протекает процесс обучения модели (рис. 7.7).



**Рис. 7.7.** С помощью TensorBoard можно наблюдать, как продвигается обучение и изменяются значения метрик

## 7.4. РАЗРАБОТКА СВОЕГО ЦИКЛА ОБУЧЕНИЯ И ОЦЕНКИ

Рабочий процесс на основе метода `fit()` обеспечивает хороший баланс между простотой и гибкостью. Именно этот подход вы будете использовать чаще всего. Однако он не предназначен для поддержки нужд исследователей глубокого обучения, даже несмотря на возможность настройки метрик, функций потерь и обратных вызовов.

В конце концов, подход с использованием метода `fit()` ориентирован исключительно на *обучение с учителем*, когда заранее известны *цели* (также называемые *метками* или *аннотациями*), связанные с входными данными, а потери вычисляются как функция этих целей и прогнозов модели. Однако не все формы машинного обучения попадают в эту категорию. В некоторых случаях нет явных целей — например, в генеративном обучении (которое мы обсудим в главе 12), в *самоконтролируемом обучении* (когда цели извлекаются из входных данных) и в *обучении с подкреплением* (когда обучение подкрепляется «вознаграждениями», что очень напоминает дрессировку собаки). Даже если вы регулярно занимаетесь

обучением с учителем, вам, как исследователю, может понадобиться добавить несколько новых опций, а для этого нужна гибкость на низком уровне.

Всякий раз, оказавшись в ситуации, когда встроенного метода `fit()` недостаточно, вам нужно будет написать свою логику обучения. Вы уже видели простые примеры низкоуровневых циклов обучения в главах 2 и 3. Напомню порядок проведения типичного цикла обучения по основным этапам.

1. Выполнить прямой проход (вычислить выходы модели) внутри `GradientTape`, чтобы получить величину потерь для текущего пакета данных.
2. Получить градиенты потерь с учетом весов модели.
3. Скорректировать веса модели, чтобы уменьшить величину потерь на текущем пакете данных.

Эти шаги повторяются для выбранного количества пакетов. Фактически именно так и действует метод `fit()`. Далее вы узнаете, как переопределить `fit()` и написать свою реализацию с нуля, что позволит вам в будущем создать любой алгоритм обучения, который только вы придумаете.

А теперь перейдем к деталям.

### 7.4.1. Обучение и прогнозирование

В примерах низкоуровневого цикла обучения, которые вы видели до сих пор, шаг 1 (прямой проход) выполнялся инструкцией `predictions = model(inputs)`, а шаг 2 (получение градиентов, вычисленных с помощью `GradientTape`) — инструкцией `gradients = tape.gradient(loss, model.weights)`. В общем случае следует учитывать две тонкости.

Некоторые слои Keras (такие как `Dropout`) во время *обучения* и во время *прогнозирования* ведут себя по-разному. Метод `call()` таких слоев принимает логический аргумент `training`. Вызов `dropout(inputs, training=True)` приведет к сбросу некоторых активаций, а вызов `dropout(inputs, training=False)` — нет. Кроме того, метод `call()` функциональных и последовательных моделей тоже поддерживает аргумент `training`. Важно не забывать передавать `training=True` при выполнении прямого прохода модели Keras! То есть прямой проход фактически должен выполняться инструкцией `predictions = model(inputs, training=True)`.

Также обратите внимание, что для получения градиентов весов модели следует использовать не `tape.gradients(loss, model.weights)`, а `tape.gradients(loss, model.trainable_weights)`. В действительности слои и модели обладают двумя видами весов, такими как:

- *обучаемые веса* — предназначены для обновления на этапе обратного распространения ошибки, чтобы минимизировать потери модели, такие как ядро и систематическая ошибка слоя `Dense`;

- *необучаемые веса* — предназначены для обновления на этапе прямого прохода слоями, которым они принадлежат. Например, если вы решите добавить в свой слой счетчик пакетов, обработанных к данному моменту, то эта информация будет храниться в необучаемом весе и после обработки каждого пакета ваш слой будет увеличивать счетчик на единицу.

Из встроенных слоев Keras необучаемые веса имеет только слой `BatchNormalization`, который мы обсудим в главе 9. Необучаемые веса нужны слою `BatchNormalization` для запоминания среднего и стандартного отклонения обрабатываемых данных, чтобы потом динамически выполнить *нормализацию признаков* (с этой идеей вы познакомились в главе 6).

Принимая во внимание эти две детали, этап обучения с учителем в конечном итоге будет выглядеть следующим образом:

```
def train_step(inputs, targets):  
    with tf.GradientTape() as tape:  
        predictions = model(inputs, training=True)  
        loss = loss_fn(targets, predictions)  
    gradients = tape.gradients(loss, model.trainable_weights)  
    optimizer.apply_gradients(zip(model.trainable_weights, gradients))
```

## 7.4.2. Низкоуровневое использование метрик

В низкоуровневом цикле обучения часто возникает необходимость использовать метрики Keras (и стандартные, и нестандартные). Вы уже познакомились с методами поддержки метрик: просто вызовите `update_state(y_true, y_pred)` для каждого пакета целей и прогнозов и `result()` для получения текущего значения метрики:

```
metric = keras.metrics.SparseCategoricalAccuracy()  
targets = [0, 1, 2]  
predictions = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]  
metric.update_state(targets, predictions)  
current_result = metric.result()  
print(f"result: {current_result:.2f}")
```

Вам также может потребоваться отслеживать среднее значение скаляра, например величины потери модели. Это можно сделать с помощью метрики `keras.metrics.Mean`:

```
values = [0, 1, 2, 3, 4]  
mean_tracker = keras.metrics.Mean()  
for value in values:  
    mean_tracker.update_state(value)  
print(f"Mean of values: {mean_tracker.result():.2f}")
```

Не забудьте вызвать `metric.reset_state()`, когда понадобится сбросить текущий результат (в начале эпохи обучения или в начале этапа оценки).

### 7.4.3. Полный цикл обучения и оценки

Давайте теперь объединим прямой проход, обратное распространение ошибки и отслеживание метрик в шаговую функцию (training step function), подобную `fit()`, которая принимает пакет данных и цели и возвращает сведения, которые будут отображаться в индикаторе выполнения `fit()`.

**Листинг 7.21.** Разработка своего цикла обучения: шаговая функция

```
model = get_mnist_model()

loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.RMSprop()
metrics = [keras.metrics.SparseCategoricalAccuracy()]
loss_tracking_metric = keras.metrics.Mean()

def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        loss = loss_fn(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))

        logs = {}
        for metric in metrics:
            metric.update_state(targets, predictions)
            logs[metric.name] = metric.result()

        loss_tracking_metric.update_state(loss)
        logs["loss"] = loss_tracking_metric.result()
    return logs
```

Важно не забыть сбросить состояние метрик в начале каждой эпохи и перед началом этапа оценки. Вот вспомогательная функция, которая сделает это.

**Листинг 7.22.** Разработка своего цикла обучения: сброс метрик

```
def reset_metrics():
    for metric in metrics:
        metric.reset_state()
    loss_tracking_metric.reset_state()
```

Теперь можно закончить реализацию цикла обучения. Обратите внимание, что здесь используется объект `tf.data.Dataset`, превращающий массив NumPy с данными в итератор, который выполняет итерации по данным пакетами размером 32.

**Листинг 7.23.** Разработка своего цикла обучения: сам цикл

```
training_dataset = tf.data.Dataset.from_tensor_slices(
    (train_images, train_labels))
training_dataset = training_dataset.batch(32)
epochs = 3
for epoch in range(epochs):
    reset_metrics()
    for inputs_batch, targets_batch in training_dataset:
        logs = train_step(inputs_batch, targets_batch)
        print(f"Results at the end of epoch {epoch}")
        for key, value in logs.items():
            print(f"...{key}: {value:.4f}")
```

Ниже приводится цикл оценки: простой цикл `for`, многократно вызывающий функцию `test_step()`, которая обрабатывает один пакет данных. Функция `test_step()` — лишь подмножество логики `train_step()`. В ней отсутствует код, обновляющий веса модели, то есть все, что связано с `GradientTape` и оптимизатором.

**Листинг 7.24.** Разработка своего цикла обучения: цикл оценки

```
def test_step(inputs, targets):
    predictions = model(inputs, training=False) ←
    loss = loss_fn(targets, predictions)           | Обратите внимание
                                                    | на аргумент training=False

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()
for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
print("Evaluation results:")
for key, value in logs.items():
    print(f"...{key}: {value:.4f}")
```

Поздравляю, вы только что реализовали свои полноценные версии функций `fit()` и `evaluate()`! Ну или почти полноценные: на самом деле `fit()` и `evaluate()` реализуют множество других возможностей, включая крупномасштабные распределенные вычисления, которые требуют немного больше работы. Они также включают некоторые важные оптимизации производительности.

Давайте рассмотрим одну из таких оптимизаций: компиляцию функции TensorFlow.

#### 7.4.4. Ускорение вычислений с помощью `tf.function`

Возможно, вы заметили, что реализованные вами циклы работают значительно медленнее, чем встроенные функции `fit()` и `evaluate()`, несмотря на то что фактически реализуют ту же логику. Причина в том, что по умолчанию код TensorFlow выполняется построчно и *немедленно*, подобно коду NumPy или обычному коду Python. Немедленное выполнение упрощает отладку, но с точки зрения производительности далеко не оптимально.

Более полезным для производительности будет скомпилировать код TensorFlow в *граф вычислений*, который можно оптимизировать глобально, что не получится сделать при построчной интерпретации кода. Синтаксис применения такой оптимизации прост: добавьте `@tf.function` к любой функции, которую нужно скомпилировать перед выполнением, как показано в следующем листинге.

**Листинг 7.25.** Добавление декоратора `@tf.function` к функции оценки

```

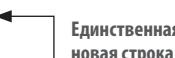
@tf.function
def test_step(inputs, targets):
    predictions = model(inputs, training=False)
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()
for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
    print("Evaluation results:")
    for key, value in logs.items():
        print(f"...{key}: {value:.4f}")

```



←

Единственная  
новая строка

В Colab время выполнения цикла оценки уменьшилось с 1,8 до 0,8 секунды. Теперь он выполняется намного быстрее!

Помните, что в процессе отладки код лучше запускать без декоратора `@tf.function`. Так проще находить и устранять ошибки. Закончив отладку, код можно ускорить, добавив декоратор `@tf.function` перед функциями, реализующими шаг обучения и шаг оценки, или любыми другими функциями, для которых важна высокая производительность.

### 7.4.5. Использование fit() с нестандартным циклом обучения

Ранее мы с нуля написали полный цикл обучения. Этот подход дает максимальную гибкость, но не только требует написать много кода, но и лишает множества удобных возможностей `fit()`, таких как обратные вызовы или встроенная поддержка распределенного обучения.

А получится ли применить свой алгоритм обучения и сохранить всю мощь встроенной логики обучения Keras? На самом деле существует золотая середина между использованием `fit()` и реализацией своего цикла обучения: можно написать свою функцию шага обучения, а все остальные задачи переложить на фреймворк.

Для этого достаточно переопределить метод `train_step()` класса `Model`, который вызывается функцией `fit()` для обработки каждого пакета данных, и использовать `fit()` как обычно, а функция будет запускать ваш алгоритм обучения.

Вот простой пример:

- создадим новый класс, наследующий класс `keras.Model`;
- переопределим метод `train_step(self, data)`, почти полностью повторив все, что мы написали выше. Теперь метод будет возвращать словарь, отображающий имена метрик (включая метрику потерь) в их текущие значения;
- реализуем свойство `metrics` для отслеживания экземпляров класса `Metric` в модели. Это позволит модели автоматически вызывать `reset_state()` для метрик в начале каждой эпохи и в начале вызова функции `evaluate()`, чтобы не делать этого вручную.

**Листинг 7.26.** Реализация своего шага обучения для использования с `fit()`

```
loss_fn = keras.losses.SparseCategoricalCrossentropy()
loss_tracker = keras.metrics.Mean(name="loss") ←
    Данный объект метрики будет
    использоваться для слежения
    за средним значением потерь
    на пакетах в ходе обучения
    и оценки

class CustomModel(keras.Model):
    def train_step(self, data): ←
        Мы переопределяем
        метод train_step
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True) ←
            loss = loss_fn(targets, predictions)
            gradients = tape.gradient(loss, model.trainable_weights)
            optimizer.apply_gradients(zip(gradients, model.trainable_weights))

            loss_tracker.update_state(loss)
            return {"loss": loss_tracker.result()} ←
                Обновить метрику потерь,
                в которой хранится среднее
                значение потерь

@property
def metrics(self): ←
    Список всех метрик, которые
    должны сбрасываться
    в исходное состояние
    в начале каждой эпохи
    return [loss_tracker] ←
        Вернуть среднее значение потерь,
        получившееся к данному моменту,
        обратившись к экземпляру
        метрики loss_tracker
```

Теперь можно создать экземпляр модели, скомпилировать ее (в данном случае мы передаем только оптимизатор, потому что потери определены вне модели) и обучить, используя `fit()` как обычно:

```
inputs = keras.Input(shape=(28 * 28,))
features = layers.Dense(512, activation="relu")(inputs)
features = layers.Dropout(0.5)(features)
outputs = layers.Dense(10, activation="softmax")(features)
model = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop())
model.fit(train_images, train_labels, epochs=3)
```

Отметим несколько важных моментов:

- данный подход можно использовать также при построении моделей с помощью функционального API — он не зависит от способа построения модели: с применением класса `Sequential`, функционального API или наследованием класса `Model`;
- при переопределении метода `train_step` не нужно использовать декоратор `@tf.function` — фреймворк сделает это автоматически.

А что насчет метрик и функции потерь, которые настраиваются с помощью `compile()`? После вызова `compile()` вы получаете доступ к:

- `self.compiled_loss` — функции потерь, переданной в вызов `compile()`;
- `self.compiled_metrics` — обертке для списка метрик, которая позволяет вызвать `self.compiled_metrics.update_state()` и обновить сразу все метрики;
- `self.metrics` — фактическому списку метрик, переданному в вызов `compile()`. Обратите внимание, что он также включает метрику, предназначенную для отслеживания потерь, подобно тому как мы делали это вручную с помощью нашей метрики `loss_tracking_metric`.

То есть мы можем написать такой класс:

```
class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = self.compiled_loss(targets, predictions) ←
                    Вычислить величину потерь
                    вызовом self.compiled_loss

            gradients = tape.gradient(loss, model.trainable_weights)
            optimizer.apply_gradients(zip(gradients, model.trainable_weights))
            self.compiled_metrics.update_state(targets, predictions) →
                    Обновить метрики модели
                    с помощью обертки
                    self.compiled_metrics
            return {m.name: m.result() for m in self.metrics} ←
                    Вернуть словарь,
                    отображающий имена метрик
                    в их текущие значения
```

Давайте опробуем его:

```
inputs = keras.Input(shape=(28 * 28,))
features = layers.Dense(512, activation="relu")(inputs)
features = layers.Dropout(0.5)(features)
outputs = layers.Dense(10, activation="softmax")(features)
model = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.SparseCategoricalCrossentropy(),
              metrics=[keras.metrics.SparseCategoricalAccuracy()])
model.fit(train_images, train_labels, epochs=3)
```

В этой главе было представлено много новой информации, зато теперь вы знаете практически все, что нужно, чтобы использовать Keras для создания почти любых моделей.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Keras предлагает целый спектр рабочих процессов, основанных на принципе *постепенного раскрытия сложности*. Все они прекрасно взаимодействуют друг с другом.
- Модели можно конструировать с помощью класса `Sequential`, функционального API или определяя свои подклассы класса `Model`. В большинстве случаев вы будете использовать функциональный API.
- Самый простой способ обучить и оценить модель — использовать методы по умолчанию `fit()` и `evaluate()`.
- 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.
- Вы можете полностью контролировать работу `fit()`, переопределив метод `train_step()`.
- Помимо функции `fit()`, можно также реализовать свой цикл обучения с нуля. Эта возможность может пригодиться исследователям, реализующим совершенно новые алгоритмы обучения.

# *Введение в глубокое обучение в технологиях компьютерного зрения*

## **В этой главе**

- ✓ Суть сверточных нейронных сетей.
- ✓ Обогащение обучающего набора данных для ослабления эффекта переобучения.
- ✓ Использование предварительно обученной сверточной нейронной сети для извлечения признаков.
- ✓ Дообучение предварительно обученной сверточной нейронной сети.

Компьютерное зрение — одна из самых первых технологий, где глубокое обучение добилось значительных успехов. Каждый день мы взаимодействуем с моделями компьютерного зрения — через Google Photos, поиск изображений Google, YouTube, видеофильтры в программном обеспечении камер, программные инструменты оптического распознавания текста и множество других приложений. Также эти модели широко используются в передовых исследованиях в сфере автоматического управления транспортными средствами, робототехники, медицинской диагностики с помощью искусственного интеллекта, автоматических систем кассового обслуживания для магазинов и даже автоматизации сельского хозяйства.

Компьютерное зрение — это предметная область, которая послужила толчком к развитию глубокого обучения в период с 2011 по 2015 год. Примерно тогда же модели глубокого обучения для компьютерного зрения — *сверточные нейронные*

сети — стали показывать удивительно хорошие результаты в состязаниях по классификации изображений. Сначала Дэн Киресан победил в двух специализированных соревнованиях (ICDAR 2011, соревнования по распознаванию китайских символов, и IJCNN 2011, соревнования по распознаванию дорожных знаков Германии). Затем произошло еще более значимое событие: осенью 2012 года группа Хинтона выиграла крупномасштабное состязание по визуальному распознаванию изображений из набора ImageNet. После этого начали появляться многообещающие результаты в других задачах компьютерного зрения.

Интересно отметить, что первых успехов было недостаточно, чтобы сделать глубокое обучение популярным, — на это потребовалось несколько лет. Сообщество исследователей технологий компьютерного зрения потратило много лет на разработку методов, не связанных с нейронными сетями, и не было готово в одночасье отказаться от них, только потому что на пороге появилось что-то новое. В 2013 и 2014 годах многие ученые в области компьютерного зрения все еще встречали идею глубокого обучения с большим скептицизмом — и только в 2016 году она наконец заняла доминирующие позиции. Помню, как в феврале 2014 года я убеждал своего бывшего профессора заняться глубоким обучением. «У этой технологии большое будущее», — говорил я. «А мне кажется, это временное поветрие», — парировал он. Но уже в 2016 году вся его лаборатория занималась глубоким обучением. Ничто не остановит идею, время которой пришло.

Данная глава знакомит со сверточными нейронными сетями (также известными как *convnets*) — разновидностью моделей глубокого обучения, почти повсеместно используемой 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).

## 8.1. 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, классифицирующей изображения рукописных 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 следующем листинге показано, как выглядит простая сверточная нейронная сеть. Это стек слоев Conv2D и MaxPooling2D. Как она действует, рассказывается чуть ниже. Мы построим модель с помощью функционального API, с которым вы познакомились в предыдущей главе.

**Листинг 8.1.** Создание небольшой сверточной нейронной сети

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

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.

Рассмотрим поближе текущую архитектуру сети.

**Листинг 8.2.** Сводная информация о сети

```
>>> model.summary()
Model: "model"
-----  
Layer (type)           Output Shape        Param #
-----  
input_1 (InputLayer)   [(None, 28, 28, 1)]  0  
conv2d (Conv2D)         (None, 26, 26, 32)    320  
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)  0  
conv2d_1 (Conv2D)       (None, 11, 11, 64)    18496  
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 64)  0  
conv2d_2 (Conv2D)       (None, 3, 3, 128)    73856  
flatten (Flatten)      (None, 1152)        0  
dense (Dense)          (None, 10)          11530  
-----  
Total params: 104,202  
Trainable params: 104,202  
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 управляетется первым аргументом, передаваемым в слои Conv2D (32, 64 или 128).

Последний слой Conv2D выдает результат с формой (3, 3, 128) — карту признаков  $3 \times 3$  со 128 каналами. Следующий шаг — передача этого результата на вход полносвязной 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. Чтобы преодолеть это несоответствие, мы преобразуем трехмерный вывод в одномерный с помощью слоя Flatten, а затем добавляем полносвязные слои Dense.

В заключение выполняется классификация по десяти категориям, поэтому последний слой имеет десять выходов и активацию softmax.

Теперь обучим сверточную сеть распознаванию цифр MNIST. Мы будем повторно брать большое количество программного кода из главы 2. Поскольку модель выполняет классификацию по десяти категориям с активацией softmax, мы используем функцию потерь категориальной перекрестной энтропии, а так как метки являются целыми числами, нам понадобится разреженная версия sparse\_categorical\_crossentropy.

#### **Листинг 8.3.** Обучение сверточной нейронной сети на данных из набора MNIST

```
(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
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Оценим модель на контрольных данных.

#### **Листинг 8.4.** Оценка сверточной сети

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"Test accuracy: {test_acc:.3f}")
Test accuracy: 0.991
```

Полносвязная сеть из главы 2 показала точность 97,8 % на контрольных данных, а простенькая сверточная нейронная сеть — 99,3 %: мы уменьшили процент ошибок на 68 % (относительно). Неплохо!

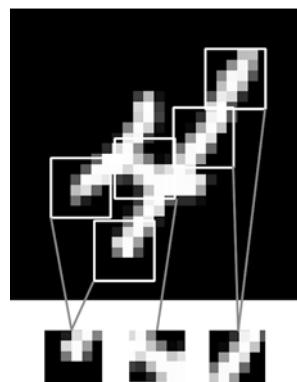
Но почему такая простая сверточная нейронная сеть работает намного лучше полносвязной модели? Чтобы ответить на этот вопрос, погрузимся в особенности работы слоев Conv2D и MaxPooling2D.

### 8.1.1. 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. 8.1): 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:

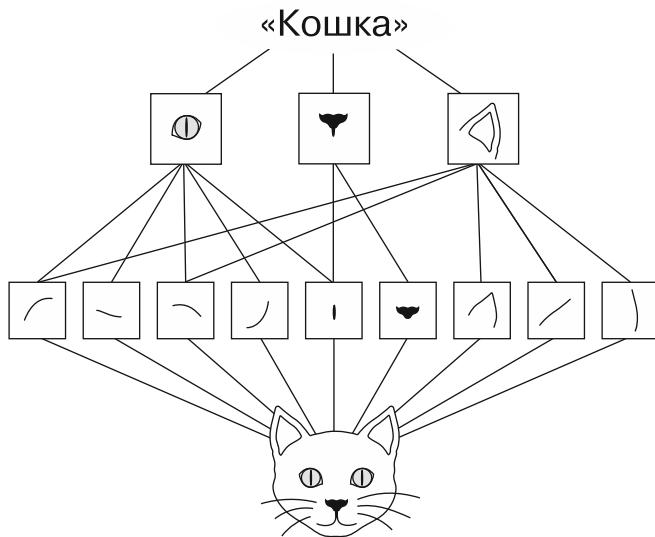
- *шаблоны, которые они изучают, являются инвариантными в отношении переноса.* После изучения определенного шаблона в правом нижнем углу картинки сверточная нейронная сеть сможет распознавать его повсюду, например в левом верхнем углу. Полносвязной сети пришлось бы изучить шаблон заново, появись он в 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.* Первый сверточный слой будет изучать небольшие локальные шаблоны, такие как края, второй — более крупные шаблоны, состоящие из признаков, возвращаемых первым слоем, и т. д. (рис. 8.2). 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. 8.1.** Изображения можно разбить на локальные шаблоны, такие как края, текстуры и т. д.

Свертка применяется к трехмерным тензорам, называемым *картами признаков*, с двумя пространственными осями (*высотой* и *шириной*), 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*. Выходная карта признаков также является трехмерным тензором: у нее есть ширина и высота. 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 конкретные аспекты входных данных: на верхнем уровне, например, фильтр может соответствовать понятию «присутствие лица на входе».



**Рис. 8.2.** Видимый мир формируется пространственными иерархиями видимых модулей: элементарные линии или текстуры объединяются в простые объекты, такие как глаза или уши, которые, в свою очередь, объединяются в понятия еще более высокого уровня, такие как «кошка»

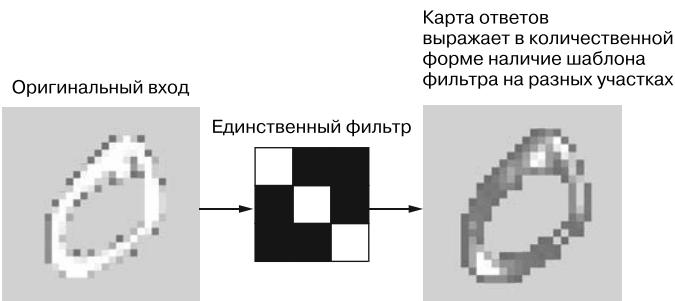
В примере MNIST первый сверточный слой принимает карту признаков размером  $(28, 28, 1)$  и выводит карту признаков размером  $(26, 26, 32)$ : он вычисляет 32 фильтра по входным данным. Каждый из этих 32 выходных каналов содержит сетку  $26 \times 26$  значений — *карту ответов* фильтра на входных данных, определяющую ответ этого шаблона фильтра для разных участков входных данных (рис. 8.3).

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* фильтра на входных данных.

Свертки определяются двумя ключевыми параметрами:

- *размером 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;

- глубиной выходной карты признаков — количеством фильтров, вычисляемых сверткой. В данном примере свертка начинается с глубины 32 и заканчивается глубиной 64.



**Рис. 8.3.** Понятие карты ответов: двумерная карта присутствия шаблона на разных участках входных данных

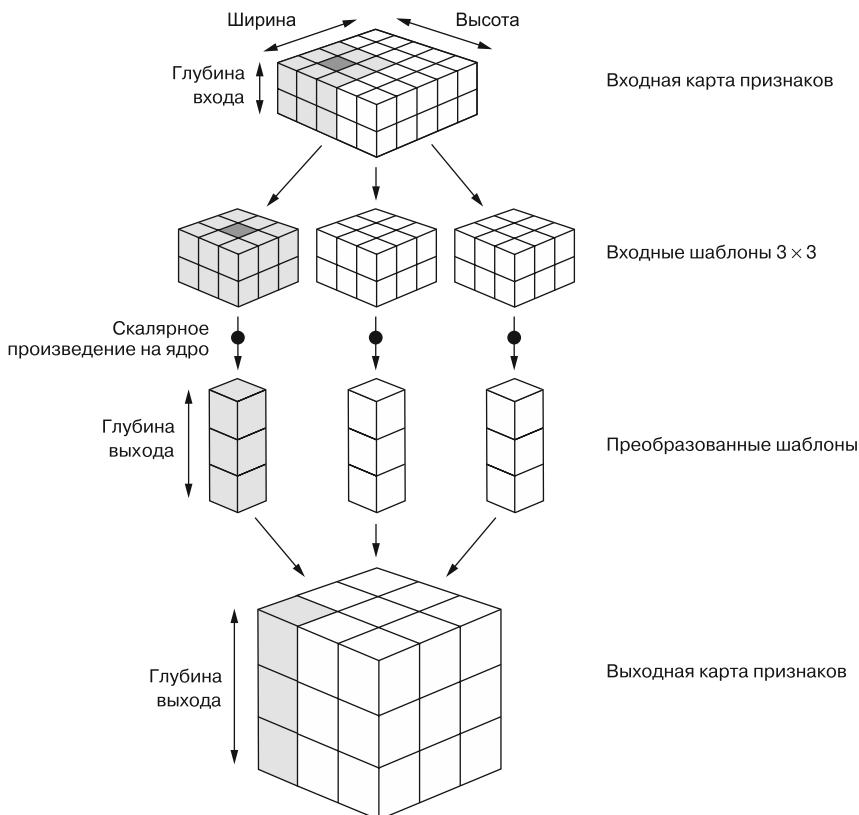
В Keras эти параметры передаются в слои `Conv2D` в первых аргументах: `Conv2D(выходная_глубина, (высота_окна, ширина_окна))`.

Свертка работает методом скользящего окна: она *двигает* окно 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 `output[i, j, :]` соответствует трехмерному шаблону `input[i-1:i+2, j-1:j+2, :]`. Полный процесс изображен на рис. 8.4.

Обратите внимание, что выходные ширина и высота могут отличаться от входных. На то есть две причины:

- эффекты границ, которые могут устраниться дополнением входной карты признаков;
- использование *шага свертки*, определение которого приводится чуть ниже.

Рассмотрим подробнее эти понятия.

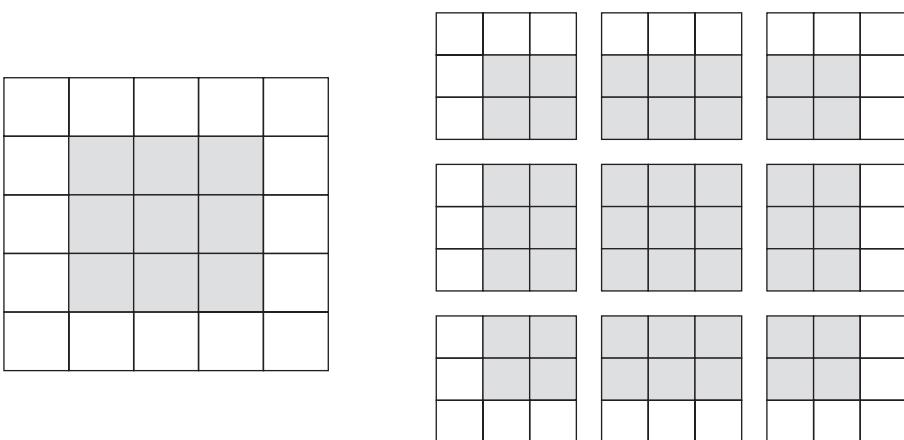


**Рис. 8.4.** Принцип действия свертки

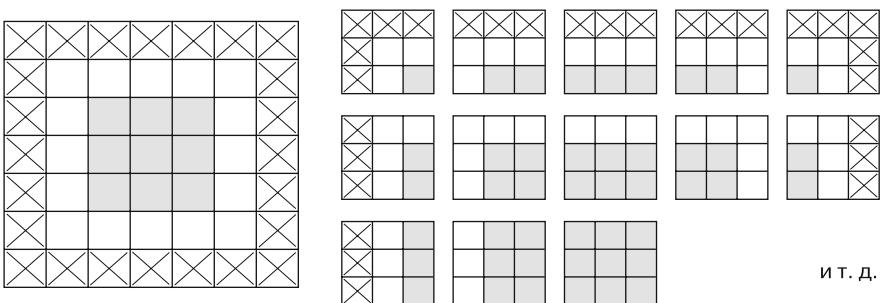
### Эффекты границ и дополнение

Перед нами карта признаков  $5 \times 5$  (всего 25 клеток). Существует всего девять клеток, в которых может находиться центр окна  $3 \times 3$ , образующих сетку  $3 \times 3$  (рис. 8.5). 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  $26 \times 26$ .

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. 8.6).



**Рис. 8.5.** Допустимые местоположения шаблонов  $3 \times 3$  во входной карте признаков  $5 \times 5$



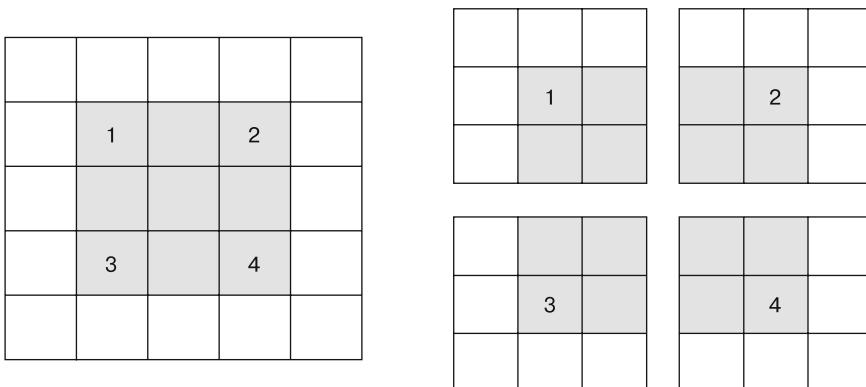
**Рис. 8.6.** Дополнение входной карты признаков  $5 \times 5$ , чтобы получить 25 шаблонов  $3 \times 3$

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 те 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. 8.7 можно видеть, как извлекаются шаблоны  $3 \times 3$  сверткой с шагом 2 из входной карты  $5 \times 5$  (без дополнения).



**Рис. 8.7.** 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 в моделях классификации, но могут пригодиться в моделях некоторых других типов — в следующей главе мы рассмотрим подробнее, как их применить.

В моделях классификации 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.

### 8.1.2. 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 таком случае будет выглядеть так, как показано в следующем листинге.

**Листинг 8.5.** Неверно структурированная сверточная сеть без слоев, выбирающих максимальное значение из соседних

```
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_no_max_pool = keras.Model(inputs=inputs, outputs=outputs)
```

Сводная информация о модели:

```
>>> model_no_max_pool.summary()
Model: "model_1"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[None, 28, 28, 1]	0
conv2d_3 (Conv2D)	(None, 26, 26, 32)	320
conv2d_4 (Conv2D)	(None, 24, 24, 64)	18496
conv2d_5 (Conv2D)	(None, 22, 22, 128)	73856
flatten_1 (Flatten)	(None, 61952)	0
dense_1 (Dense)	(None, 10)	619530
<hr/>		
Total params:	712,202	
Trainable params:	712,202	
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 классификатора цифр (попробуйте распознать цифру, посмотрев на нее через окна  $7 \times 7$  пикселей!). Нам нужно, чтобы признаки, полученные от последнего сверточного слоя, содержали информацию о совокупности исходных данных;
- заключительная карта признаков имеет  $22 \times 22 \times 128 = 61\,952$  коэффициента на образец. Это очень большое число. Если бы вы решили сделать ее плоской, чтобы наложить сверху слой Dense размером 10, данный слой имел бы полмиллиона 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, что операция выбора максимального значения не единственный способ уменьшения разрешения. Как вы уже знаете, в предыдущих сверточных слоях можно также использовать шаг свертки. Кроме того, вместо выбора максимального значения можно 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 (отсюда и название — *карта признаков*), соответственно, *максимальное присутствие* признаков намного информативнее *среднего присутствия*. Поэтому для снижения разрешения более разумно 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 вариантам применения.

## 8.2. 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 5000 изображений (2500 кошек, 2500 собак). Мы будем использовать 2000 изображений для обучения, 1000 для проверки и 2000 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 около 70 %. С этого момента начнет проявляться эффект переобучения. Затем вашему вниманию будет представлен эффективный способ уменьшения степени переобучения в распознавании образов — *обогащение данных* (data augmentation). С его помощью мы повысим точность классификации до 80–85 %.

В следующем разделе мы рассмотрим еще два важных приема глубокого обучения на небольших наборах данных: *выделение признаков с использованием предварительно обученной модели* (поможет поднять точность до 97,5 %) и *дообучение предварительно обученной сети* (поможет достичь окончательной точности 98,5 %). 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.

### 8.2.1. Целесообразность глубокого обучения для решения задач с небольшими наборами данных

Понятие «достаточное количество образцов» весьма относительно — в первую очередь, относительно размера и глубины обучаемой модели. Нельзя обучить сверточную нейронную сеть решению сложной задачи на нескольких десятках образцов, а вот нескольких сотен вполне может хватить, если модель невелика

и хорошо 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 применяться как основа для создания очень мощных моделей распознавания образов на небольших объемах данных. Возможность повторного использования признаков — одна из самых замечательных особенностей глубокого обучения. Мы исследуем ее в следующем разделе.

А пока начнем с получения данных.

## 8.2.2. 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 года, 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) (вам потребуется создать учетную запись на сайте Kaggle, если у вас ее еще нет, но не волнуйтесь, процесс регистрации очень прост). Также есть возможность взять Kaggle API для загрузки набора данных в Colab, как описывается во врезке «Загрузка набора данных из Kaggle в Google Colaboratory».

Этот набор содержит изображения в формате JPEG со средним разрешением. На рис. 8.8 показано несколько примеров.

Неудивительно, что соревнование по классификации изображений кошек и собак на сайте Kaggle в 2013 году выиграли участники, использовавшие сверточные нейронные сети. Лучшие результаты достигали точности 95 %. В нашем 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.

### ЗАГРУЗКА НАБОРА ДАННЫХ ИЗ KAGGLE В GOOGLE COLABORATORY

Kaggle поддерживает простой в использовании API для загрузки наборов данных программным способом. Вы можете использовать его, например, чтобы загрузить набор данных Dogs vs. Cats в блокнот Colab. API доступен в виде пакета `kaggle`, предустановленного в Colab. Чтобы загрузить этот набор данных, достаточно выполнить следующую команду в ячейке Colab:

```
!kaggle competitions download -c dogs-vs-cats
```

Однако этот API доступен только пользователям Kaggle, поэтому, прежде чем приступить к выполнению предыдущей команды, следует аутентифицировать себя. Пакет `kaggle` будет искать ваши учетные данные в JSON-файле `~/.kaggle/kaggle.json`. Давайте создадим этот файл.

Сначала создайте ключ Kaggle API и загрузите его на свой компьютер. Для этого войдите на сайт Kaggle в браузере, зарегистрируйтесь и перейдите на страницу My Account (Моя учетная запись). В настройках учетной записи вы найдете раздел API. Нажмите кнопку Create New API Token (Создать новый токен API), чтобы сгенерировать файл ключа `kaggle.json` и загрузить его на свой компьютер.

Затем перейдите в блокнот Colab и выгрузите JSON-файл с ключом API в сеанс Colab, выполнив следующий код в ячейке блокнота:

```
from google.colab import files  
files.upload()
```

После этого появится кнопка Choose Files (Выбрать файлы). Щелкните на ней и выберите только что загруженный файл `kaggle.json`. После этого файл будет выгружен в локальную среду выполнения Colab.

Наконец, создайте папку `~/.kaggle` (`mkdir ~/.kaggle`) и скопируйте в нее файл ключа (`cp kaggle.json ~/.kaggle/`). Для безопасности сделайте файл доступным для чтения только текущему пользователю (`chmod 600`):

```
!mkdir ~/.kaggle  
!cp kaggle.json ~/.kaggle/  
!chmod 600 ~/.kaggle/kaggle.json
```

Теперь можно загрузить данные, которые мы будем использовать:

```
!kaggle competitions download -c dogs-vs-cats
```

При первой попытке загрузки данных может возникнуть ошибка 403 Forbidden. Это объясняется необходимостью принять условия, связанные с набором данных, прежде чем загрузить их, — перейдите на страницу [www.kaggle.com/c/dogs-vs-cats/rules](http://www.kaggle.com/c/dogs-vs-cats/rules) (предварительно зарегистрировавшись со своими учетными данными Kaggle) и нажмите кнопку I Understand and Accept (Я понимаю и принимаю). Это нужно сделать только один раз.

Кроме того, обучающие данные находятся в сжатом архиве `train.zip`. Распакуйте его (`unzip`) в режиме без вывода сообщений (`-qq`):

```
!unzip -qq train.zip
```



**Рис. 8.8.** Примеры изображений из набора Dogs vs. Cats. Размеры не были изменены: изображения имеют разные размеры, ракурсы съемки и т. д.

Данный 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 образцов каждого класса, проверочный набор с 500 образцами каждого класса и контрольный набор с 1000 образцов каждого класса. «Зачем?» — спросите вы. Дело в том, что многие наборы изобразительных данных, с которыми вы столкнетесь в своей карьере, содержат всего несколько тысяч образцов, а не десятки тысяч. Наличие большего количества данных усложнило бы задачу, поэтому рекомендуется проводить обучение с использованием небольшого набора данных.

После разделения набора данных должна получиться следующая структура каталогов:

```
cats_vs_dogs_small/
...train/
.....cat/ ← Содержит 1000 изображений кошек
.....dog/ ← Содержит 1000 изображений собак
...validation/
.....cat/ ← Содержит 500 изображений кошек
.....dog/ ← Содержит 500 изображений собак
...test/
.....cat/ ← Содержит 1000 изображений кошек
.....dog/ ← Содержит 1000 изображений собак
```

Все необходимое можно сделать парой вызовов `shutil`.

**Листинг 8.6.** Копирование изображений в обучающий, проверочный и контрольный каталоги

```

import os, shutil, pathlib
original_dir = pathlib.Path("train")
new_base_dir = pathlib.Path("cats_vs_dogs_small")
def make_subset(subset_name, start_index, end_index):
    for category in ("cat", "dog"):
        dir = new_base_dir / subset_name / category
        os.makedirs(dir)
        fnames = [f"{category}.{i}.jpg"
                  for i in range(start_index, end_index)]
        for fname in fnames:
            shutil.copyfile(src=original_dir / fname,
                            dst=dir / fname)

make_subset("train", start_index=0, end_index=1000)
make_subset("validation", start_index=1000, end_index=1500)
make_subset("test", start_index=1500, end_index=2500)

```

Путь к каталогу с распакованным исходным набором данных

Каталог для сохранения выделенного небольшого набора

Создать обучающий поднабор, включающий первые 1000 изображений из каждой категории

Создать проверочный поднабор, включающий следующие 500 изображений из каждой категории

Создать контрольный поднабор, включающий следующие 1000 изображений из каждой категории

Вспомогательная функция для копирования изображений кошек (и собак) с индексами от start\_index до end\_index в подкаталог new\_base\_dir/{subset\_name}/cat (и/dog). В параметре subset\_name будет передаваться строка "train", "validation" или "test"

Теперь у нас имеется 2000 обучающих, 1000 проверочных и 2000 контрольных изображений. Каждый поднабор содержит одинаковое количество образцов каждого класса: это сбалансированная задача бинарной классификации, соответственно, мерой успеха может служить точность классификации.

### 8.2.3. Конструирование сети

Далее мы вновь реализуем модель с той же общей структурой, что и в первом примере: 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, добавив еще одну пару слоев Conv2D и `MaxPooling2D`. Это увеличит емкость модели и обеспечит дополнительное снижение размеров карт признаков, чтобы они не оказались слишком большими, когда достигнут слоя `Flatten`. Учитывая, что мы начнем с входов, имеющих размер  $180 \times 180$  (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 256), а их размеры — уменьшаться (со  $180 \times 180$  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` размером 1 и функцией активации `sigmoid`). Этот признак будет представлять вероятность принадлежности рассматриваемого изображения одному из двух классов.

И еще одно небольшое отличие: модель будет начинаться со слоя `Rescaling`, преобразующего входные данные (значения которых изначально находятся в диапазоне [0, 255]) в диапазон [0, 1].

**Листинг 8.7.** Создание небольшой сверточной нейронной сети для классификации изображений кошек и собак

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1./255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Модель принимает изображения в формате RGB размерами 180 × 180

Привести входные данные к диапазону [0, 1] делением на 255

Посмотрим, как изменяются размеры карт признаков с каждым последующим слоем:

```
>>> model.summary()
Model: "model_2"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_3 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d_6 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_7 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_3 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_8 (Conv2D)	(None, 41, 41, 128)	73856

max_pooling2d_4 (MaxPooling2	(None, 20, 20, 128)	0
conv2d_9 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_5 (MaxPooling2	(None, 9, 9, 256)	0
conv2d_10 (Conv2D)	(None, 7, 7, 256)	590080
flatten_2 (Flatten)	(None, 12544)	0
dense_2 (Dense)	(None, 1)	12545
<hr/> <hr/> <hr/>		
Total params:	991,041	
Trainable params:	991,041	
Non-trainable params:	0	

На этапе компиляции, как обычно, используем оптимизатор RMSprop. Так как модель заканчивается единственным сигмоидным выходом, используем функцию потерь `binary_crossentropy` (для напоминания: в табл. 6.1 приводится шпаргалка по использованию разных функций потерь в разных ситуациях).

#### Листинг 8.8. Настройка модели для обучения

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

### 8.2.4. Предварительная обработка данных

Как вы уже знаете, перед передачей в модель данные должны быть преобразованы в тензоры с вещественными числами. В настоящее время данные хранятся в виде файлов JPEG, поэтому их нужно подготовить, выполнив следующие шаги.

1. Прочитать файлы с изображениями.
2. Декодировать содержимое из формата JPEG в матрицы пикселей RGB.
3. Преобразовать их в тензоры с вещественными числами.
4. Привести к единому размеру (в нашем случае  $180 \times 180$ ).
5. Организовать в пакеты (мы будем использовать пакеты по 32 изображения в каждом).

Этот порядок действий может показаться немного сложным, но, к счастью, в Keras имеются утилиты, способные выполнить его автоматически. В частности, вспомогательная функция `image_dataset_from_directory()`, которая позволит быстро настроить конвейер обработки для автоматического преобразования файлов с изображениями в пакеты готовых тензоров. Именно ее мы и возьмем.

Вызов `image_dataset_from_directory(directory)` сначала составит список подкаталогов в каталоге `directory` и предположит, что каждый содержит изображения, принадлежащие одному из классов. Затем проиндексирует файлы изображений в каждом подкаталоге и, наконец, создаст и вернет объект `tf.data.Dataset`, подготовленный для чтения файлов, перемешивания, преобразования в тензоры, приведения к общему размеру и упаковки в пакеты.

**Листинг 8.9.** Чтение изображений с помощью функции `image_dataset_from_directory`

```
from tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```

### ОБЪЕКТ DATASET В TENSORFLOW

В библиотеке TensorFlow имеется модуль `tf.data`, позволяющий создавать эффективные конвейеры ввода для моделей машинного обучения. Его основу составляет класс `tf.data.Dataset`.

Объект `Dataset` — это итератор: его можно использовать в цикле `for`. Обычно он возвращает пакеты входных данных и меток. `Dataset` можно также передавать непосредственно в вызов метода `fit()` модели Keras.

Класс `Dataset` предлагает множество важных функций, избавляя вас от сложностей их реализации вручную: в частности, асинхронную предварительную выборку данных (подготовку следующего пакета данных, пока предыдущий обрабатывается моделью, что обеспечивает непрерывность потока выполнения).

Класс `Dataset` также поддерживает API в функциональном стиле для изменения наборов данных. Вот короткий пример. Создадим экземпляр `Dataset` из массива NumPy случайных чисел с набором из 1000 образцов, каждый из которых является вектором, содержащим 16 чисел:

```
import numpy as np
import tensorflow as tf
random_numbers = np.random.normal(size=(1000, 16))
dataset = tf.data.Dataset.from_tensor_slices(random_numbers)
```

Метод класса `from_tensor_slices()` можно использовать для создания экземпляра `Dataset` из массива NumPy, а также из кортежа или словаря с массивами NumPy

По умолчанию наш набор данных `dataset` возвращает образцы поодиночке:

```
>>> for i, element in enumerate(dataset):
>>>     print(element.shape)
>>>     if i >= 2:
>>>         break
(16,)
(16,)
(16,)
```

Но если добавить вызов метода `.batch()`, образцы будут возвращаться пакетами:

```
>>> batched_dataset = dataset.batch(32)
>>> for i, element in enumerate(batched_dataset):
>>>     print(element.shape)
>>>     if i >= 2:
>>>         break
(32, 16)
(32, 16)
(32, 16)
```

Более того, в вашем распоряжении есть целый ряд удобных методов для работы с набором данных, таких как:

- `.shuffle(buffer_size)` — перемешивает элементы в буфере;
- `.prefetch(buffer_size)` — выполняет предварительную выборку буфера элементов в памяти GPU для большей эффективности;
- `.map(callable)` — применяет произвольное преобразование к каждому элементу в наборе данных (функцию `callable`, которая должна принимать один элемент из набора данных).

Особенно часто вам будет нужен метод `.map()`. Вот пример его использования для преобразования элементов с формой `(16, )` в нашем наборе данных в элементы с формой `(4, 4)`:

```
>>> reshaped_dataset = dataset.map(lambda x: tf.reshape(x, (4, 4)))
>>> for i, element in enumerate(reshaped_dataset):
>>>     print(element.shape)
>>>     if i >= 2:
>>>         break
(4, 4)
(4, 4)
(4, 4)
```

Далее в этой главе вы увидите еще несколько примеров использования метода `map()`.

Рассмотрим вывод одного из таких объектов `Dataset`: он возвращает пакеты изображений  $180 \times 180$  в формате RGB (с формой `(32, 180, 180, 3)`) и целочисленные

метки (с формой `(32, )`). В каждом пакете содержится 32 образца (в соответствии с параметром `batch_size`).

**Листинг 8.10.** Вывод форм данных и меток, возвращаемых объектом `Dataset`

```
>>> for data_batch, labels_batch in train_dataset:
>>>     print("data batch shape:", data_batch.shape)
>>>     print("labels batch shape:", labels_batch.shape)
>>>     break
data batch shape: (32, 180, 180, 3)
labels batch shape: (32,)
```

Давайте обучим модель на нашем наборе данных и возьмем аргумент `validation_data` метода `fit()` для наблюдения за изменением метрик на этапе проверки с использованием отдельного объекта `Dataset`.

Кроме того, применим обратный вызов `ModelCheckpoint`, сохраняющий модель после каждой эпохи, и настроим его, указав путь к файлу для сохранения и аргументы `save_best_only=True` и `monitor="val_loss"`: с ними обратный вызов будет сохранять новый файл с весами модели (перезаписывая любой предыдущий), только если текущее значение метрики `val_loss` окажется ниже, чем когда-либо раньше во время обучения. Благодаря этому сохраненный файл всегда будет содержать состояние модели, соответствующее наиболее эффективной эпохе обучения с точки зрения величины потерь на этапе проверки, и нам не придется повторно обучать новую модель с меньшим количеством эпох, если будет достигнут эффект переобучения: мы сможем просто загрузить сохраненный файл.

**Листинг 8.11.** Обучение модели с использованием объекта `Dataset`

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]

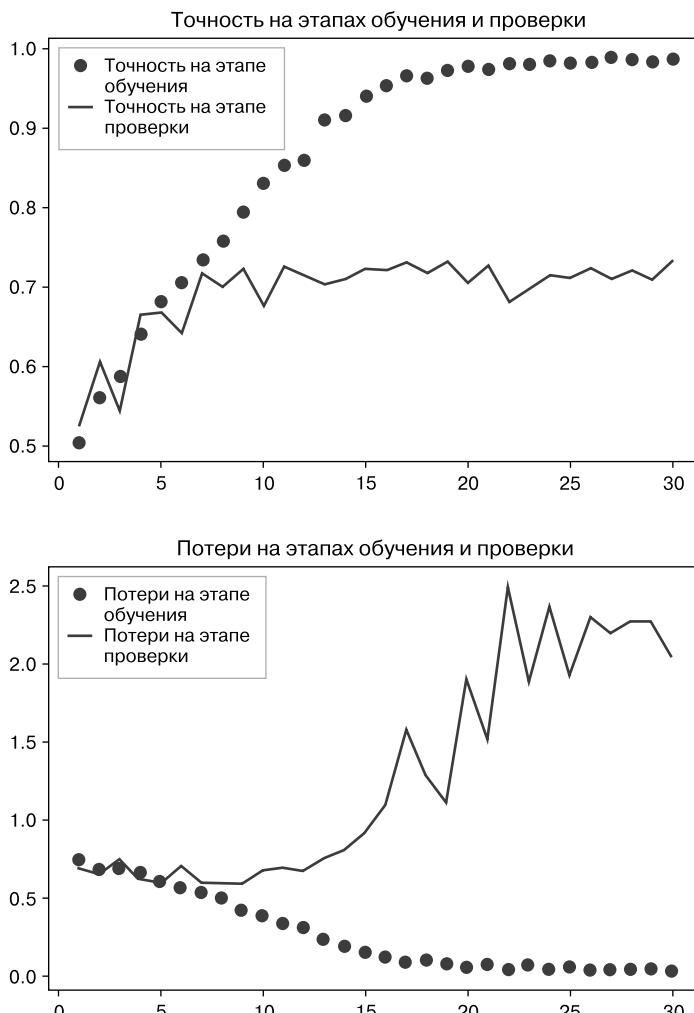
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

Создадим графики изменения точности и потерь модели на обучающих и проверочных данных в процессе обучения (рис. 8.9).

**Листинг 8.12.** Формирование графиков изменения потерь и точности в процессе обучения

```
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
```

```
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Точность на этапе обучения")
plt.plot(epochs, val_accuracy, "b", label="Точность на этапе проверки")
plt.title("Точность на этапах обучения и проверки")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Потери на этапе обучения")
plt.plot(epochs, val_loss, "b", label="Потери на этапе проверки")
plt.title("Потери на этапах обучения и проверки")
plt.legend()
plt.show()
```



**Рис. 8.9.** Графики изменения метрик на этапе проверки при обучении простой сверточной сети

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 75 %. Потери на этапе проверки достигают минимума всего после пяти эпох и затем замирают, а потери на этапе обучения продолжают линейно уменьшаться.

Проверим точность модели на контрольных данных. Для этого загрузим модель, сохраненную в файл до того, как начал проявляться эффект переобучения.

#### **Листинг 8.13.** Оценка модели на контрольном наборе

```
test_model = keras.models.load_model("convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Мы получили точность 69,5 %. (Первоначальные веса нейронной сети инициализируются случайными числами, поэтому вы можете получить немного другой результат, с разницей в пределах одного процента.)

Поскольку у нас относительно немного обучающих образцов (2000), переобучение становится проблемой номер один. Вы уже знаете несколько методов, помогающих смягчить ее, таких как прореживание и сокращение весов (L2-регуляризация). Теперь вы познакомились еще с одним способом, характерным для распознавания образов и используемым почти повсеместно при обработке изображений с применением моделей глубокого обучения: способом *обогащения данных* (data augmentation).

### **8.2.5. Обогащение 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 обогащения 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 подобное в Keras можно путем добавления нескольких *слоев обогащения данных* в начале модели. Начнем с простого примера. Приведенная далее последовательная модель применяет несколько случайных преобразований к изображениям. Мы добавим ее в нашу модель прямо перед слоем **Rescaling**.

**Листинг 8.14.** Определение этапа обогащения данных для добавления в модель

```
data_augmentation = keras.Sequential(  
    [  
        layers.RandomFlip("horizontal"),  
        layers.RandomRotation(0.1),  
        layers.RandomZoom(0.2),  
    ]  
)
```

Здесь представлена лишь часть возможных вариантов (полный список вы найдете в документации к фреймворку Keras). Давайте быстро пробежимся по этому коду:

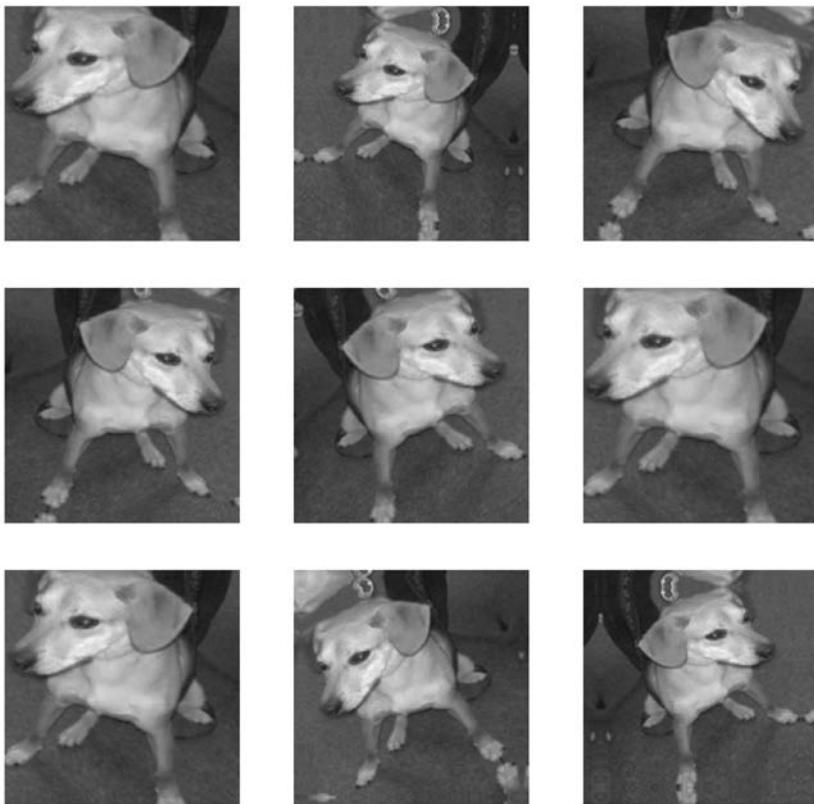
- `RandomFlip("horizontal")` — переворачивает по горизонтали 50 % случайно выбранных изображений;
- `RandomRotation(0.1)` — поворачивает входные изображения на случайный угол в диапазоне  $[-10\%, +10\%]$  (параметр определяет долю полной окружности — в градусах заданный здесь диапазон составит  $[-36, +36]$ );
- `RandomZoom(0.2)` — случайным образом изменяет масштаб изображения, в данном случае в диапазоне  $[-20\%, +20\%]$ .

Давайте посмотрим, как выглядят дополнительные изображения (рис. 8.10).

**Листинг 8.15.** Отображение некоторых дополнительных обучающих изображений

```
take(N) позволяет выбрать только N пакетов из набора  
данных. Этот метод действует подобно инструкции  
break, выполняемой циклом после N-го пакета  
  
plt.figure(figsize=(10, 10))  
for images, _ in train_dataset.take(1): ← Применить этап  
    for i in range(9): ← обогащения к пакету  
        augmented_images = data_augmentation(images) ← изображений  
        ax = plt.subplot(3, 3, i + 1)  
        plt.imshow(augmented_images[0].numpy().astype("uint8")) ←  
        plt.axis("off")  
    Вывести первое изображение в выходном пакете. Во всех девяти  
    итерациях будут получены дополнительные варианты, полученные  
    обогащением одного и того же изображения
```

Если обучить новую модель с использованием этих обогащенных 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 полно связанным классификатором, который будет выполнять прореживание.



**Рис. 8.10.** Варианты изображения с собакой, полученные применением случайных преобразований

И последнее, что следует знать о слоях обогащения изображений случайными преобразованиями: так же как Dropout, они неактивны на этапе прогнозирования (когда вызывается метод `predict()` или `evaluate()`). Во время оценки модель будет вести себя так, как если бы мы не задействовали прореживание и обогащение данных.

**Листинг 8.16.** Определение новой сверточной нейронной сети с обогащением и прореживанием

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
```

```
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

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

Теперь обучим модель, используя обогащение и прореживание данных. Поскольку ожидается, что переобучение произойдет намного позже, зададим в три раза больше эпох обучения — 100.

#### Листинг 8.17. Обучение регуляризованной модели

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

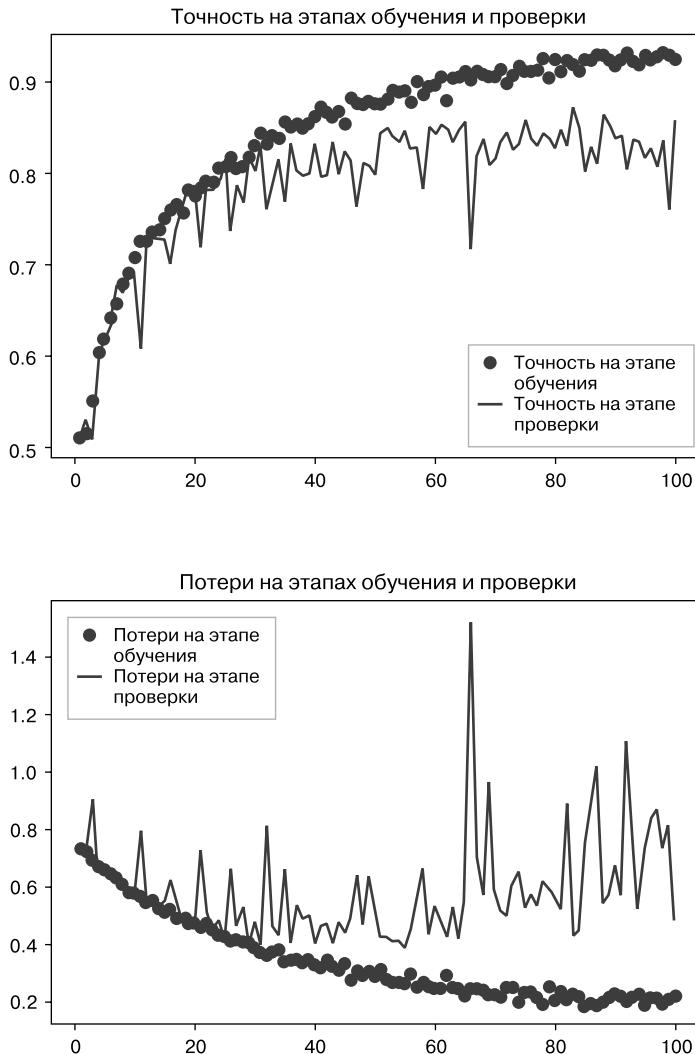
И снова выведем графики с результатами (рис. 8.11). Благодаря обогащению и прореживанию данных переобучение наступило намного позже — в районе 60–70-й эпохи (сравните с десятью эпохами в оригинальной модели). Точность на этапе проверки остановилась в районе 80–85 % — существенное улучшение по сравнению с первой попыткой.

Теперь проверим точность на контрольных данных.

#### Листинг 8.18. Оценка модели на контрольном наборе данных

```
test_model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Точность на контрольных данных составила 83,5 %. Уже неплохо! Если для экспериментов вы используете Colab, то не забудьте загрузить сохраненный файл (`convnet_from_scratch_with_augmentation.keras`): мы возьмем его для экспериментов в следующей главе.



**Рис. 8.11.** Графики изменения метрик на этапе проверки при обучении модели с обогащением данных

Используя дополнительные методы регуляризации и настраивая параметры сети (например, число фильтров на сверточный слой или число слоев в сети), можно добиться еще более высокой точности: примерно 90 %. 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 нашей задачи заключается в использовании предварительно обученной модели, но об этом мы поговорим в следующих двух разделах.

### 8.3. ИСПОЛЬЗОВАНИЕ ПРЕДВАРИТЕЛЬНО ОБУЧЕННОЙ МОДЕЛИ

Типичным и эффективным подходом к глубокому обучению на небольших наборах изображений является использование предварительно обученной модели. *Предварительно обученная модель* — это сохраненная модель, прежде обученная на большом наборе данных, обычно в рамках масштабной задачи классификации изображений. Если исходный набор данных достаточно велик и достаточно обобщен, то пространственная иерархия признаков, изученных моделью, может с успехом выступать в роли обобщенной модели видимого мира и использоваться во многих задачах компьютерного зрения, даже если новые задачи будут связаны с совершенно иными классами, отличными от встречавшихся в оригинальной задаче. Например, можно обучить сеть на изображениях из ImageNet (где подавляющее большинство классов — животные и бытовые предметы) и затем использовать ее 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.

Для нашего случая мы возьмем за основу сверточную нейронную сеть, обученную на наборе ImageNet (1,4 миллиона изображений, классифицированных по 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 задачей классификации изображений кошек и собак.

Мы воспользуемся архитектурой VGG16, разработанной Кареном Симоняном и Эндрю Циссерманом в 2014 году<sup>1</sup>. Хотя это довольно старая модель, сильно отставшая от актуального уровня и к тому же более тяжелая, чем многие современные варианты, я выбрал ее, потому что ее архитектура похожа на примеры, представленные выше в данной книге, 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, Xception и т. д.; но со временем вы привыкнете к ним, потому что они часто будут попадаться вам на 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). Начнем с первого.

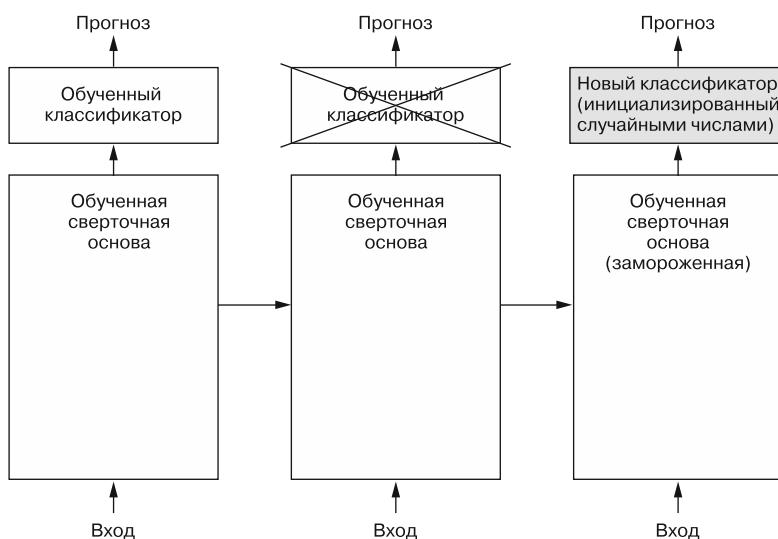
---

<sup>1</sup> Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition // arXiv, 2014, <https://arxiv.org/abs/1409.1556>.

### 8.3.1. Выделение признаков

Выделение признаков заключается в использовании представлений, изученных предварительно обученной моделью, для выделения признаков из новых образцов. Эти признаки 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. 8.12).



**Рис. 8.12.** Замена классификаторов при использовании одной и той же сверточной основы

Почему повторно используется только сверточная основа? Нельзя ли снова взять полносвязный классификатор? В общем случае этого следует избегать. Причина в том, что представления, полученные сверточной основой, обычно более 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 полносвязных 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, полносвязные признаки почти бесполезны.

Отмечу 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 получится 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 снова 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. Давайте 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 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;
- ResNet;
- MobileNet;
- EfficientNet;
- DenseNet и др.

Создадим экземпляр модели VGG16.

#### Листинг 8.19. Создание экземпляра сверточной основы VGG16

```
conv_base = keras.applications.vgg16.VGG16(  
    weights="imagenet",  
    include_top=False,  
    input_shape=(180, 180, 3))
```

Здесь конструктору передаются три аргумента:

- аргумент `weights` определяет источник весов для инициализации модели;
- аргумент `include_top` определяет необходимость подключения к сети полно связного классификатора. По умолчанию полно связный классификатор определяет принадлежность изображения к 1000 классов. Так как мы намереваемся использовать свой полно связный 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. В нашем примере мы передаем его, чтобы иметь возможность видеть (в следующей сводке), как уменьшается размер карт признаков с каждым новым слоем свертки и объединения.

Далее приводится информация о сверточной основе VGG16. Она напоминает простые сверточные нейронные сети, уже знакомые вам:

```
>>> conv_base.summary()
Model: "vgg16"
```

Layer (type)	Output Shape	Param #
input_19 (InputLayer)	[(None, 180, 180, 3)] 0	
block1_conv1 (Conv2D)	(None, 180, 180, 64)	1792
block1_conv2 (Conv2D)	(None, 180, 180, 64)	36928
block1_pool (MaxPooling2D)	(None, 90, 90, 64)	0
block2_conv1 (Conv2D)	(None, 90, 90, 128)	73856
block2_conv2 (Conv2D)	(None, 90, 90, 128)	147584
block2_pool (MaxPooling2D)	(None, 45, 45, 128)	0
block3_conv1 (Conv2D)	(None, 45, 45, 256)	295168
block3_conv2 (Conv2D)	(None, 45, 45, 256)	590080
block3_conv3 (Conv2D)	(None, 45, 45, 256)	590080
block3_pool (MaxPooling2D)	(None, 22, 22, 256)	0
block4_conv1 (Conv2D)	(None, 22, 22, 512)	1180160
block4_conv2 (Conv2D)	(None, 22, 22, 512)	2359808

```
block4_conv3 (Conv2D)           (None, 22, 22, 512)  2359808
block4_pool (MaxPooling2D)      (None, 11, 11, 512)   0
block5_conv1 (Conv2D)           (None, 11, 11, 512)  2359808
block5_conv2 (Conv2D)           (None, 11, 11, 512)  2359808
block5_conv3 (Conv2D)           (None, 11, 11, 512)  2359808
block5_pool (MaxPooling2D)      (None, 5, 5, 512)    0
=====
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

---

Заключительная карта признаков имеет форму  $(5, 5, 512)$ . Поверх нее мы положим полносвязный 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 главе 4 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 обогащения данных;
- дополнить имеющуюся модель (`conv_base`) слоями `Dense` и пропустить все входные данные. Этот путь позволяет использовать обогащение 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 обогащения данных**

Сначала выделим признаки в массив NumPy, вызвав метод `predict()` модели `conv_base` для обучающих, проверочных и контрольных данных.

Для этого выполним обход наших наборов данных и выделим признаки VGG16.

**Листинг 8.20.** Выделение признаков VGG16 и соответствующих меток

```
import numpy as np

def get_features_and_labels(dataset):
    all_features = []
    all_labels = []
    for images, labels in dataset:
        preprocessed_images = keras.applications.vgg16.preprocess_input(images)
        features = conv_base.predict(preprocessed_images)
        all_features.append(features)
        all_labels.append(labels)
    return np.concatenate(all_features), np.concatenate(all_labels)

train_features, train_labels = get_features_and_labels(train_dataset)
val_features, val_labels = get_features_and_labels(validation_dataset)
test_features, test_labels = get_features_and_labels(test_dataset)
```

Важно отметить, что `predict()` принимает только изображения, без меток, а наш объект набора данных выдает пакеты, содержащие как изображения, так и их метки. Более того, модель VGG16 принимает данные, предварительно обработанные с помощью функции `keras.applications.vgg16.preprocess_input`, которая приводит значения пикселей в соответствующий диапазон.

В настоящий момент выделенные признаки имеют форму (образцы, 5, 5, 512):

```
>>> train_features.shape
(2000, 5, 5, 512)
```

Теперь можно определить полносвязный классификатор (обратите внимание, что для регуляризации здесь используется прием прореживания) и обучить его на только что записанных данных и метках.

**Листинг 8.21.** Определение и обучение полносвязного классификатора

```
inputs = keras.Input(shape=(5, 5, 512))
x = layers.Flatten()(inputs)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)

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

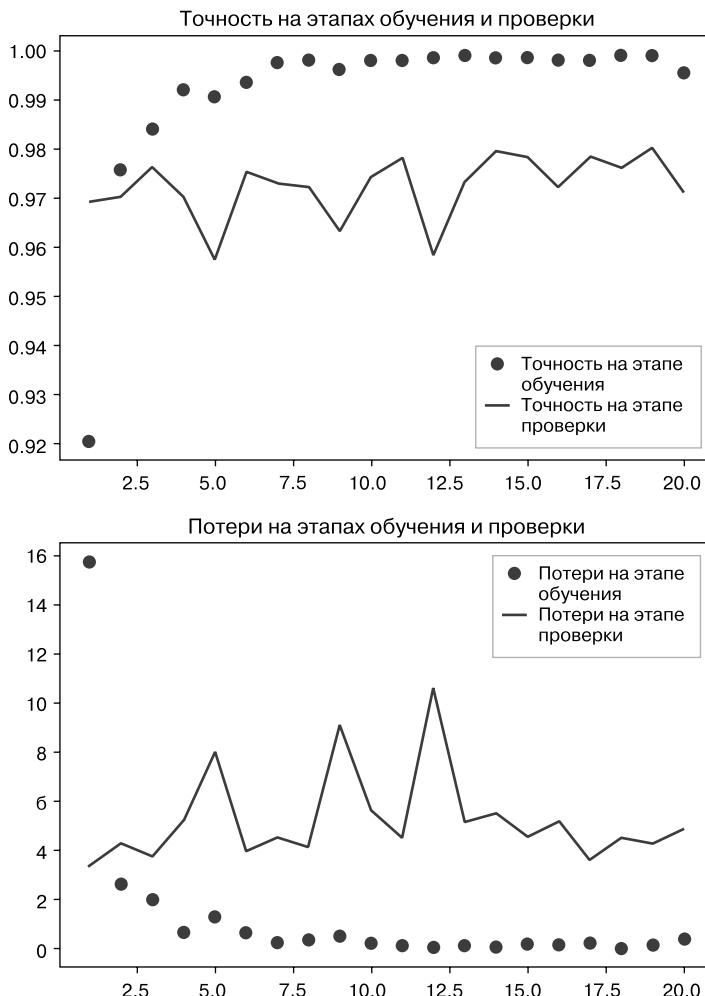
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction.keras",
        save_best_only=True,
        monitor="val_loss")
]
```

Обратите внимание, что перед передачей признаков в слой Dense они обрабатываются слоем Flatten

```
history = model.fit(
    train_features, train_labels,
    epochs=20,
    validation_data=(val_features, val_labels),
    callbacks=callbacks)
```

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. 8.13).



**Рис. 8.13.** Изменение метрик на этапах проверки и обучения для простого извлечения признаков

**Листинг 8.22.** Построение графиков с результатами

```
import matplotlib.pyplot as plt
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, "bo", label="Точность на этапе обучения")
plt.plot(epochs, val_acc, "b", label="Точность на этапе проверки")
plt.title("Точность на этапах обучения и проверки")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Потери на этапе обучения")
plt.plot(epochs, val_loss, "b", label="Потери на этапе проверки")
plt.title("Потери на этапах обучения и проверки")
plt.legend()
plt.show()
```

Мы достигли точности, близкой к 97 %, — более высокой, чем в предыдущем разделе, где обучали небольшую модель с нуля. Впрочем, это не совсем справедливое сравнение: ImageNet содержит много изображений собак и кошек, а это означает, что взятая нами предварительно обученная модель уже обладает знаниями, необходимыми для решения поставленной задачи. Но так бывает не всегда, когда используются предварительно выученные признаки.

Кроме того, графики показывают, что почти с самого начала стал проявляться эффект переобучения, несмотря на выбор довольно большого коэффициента прореживания. Это объясняется тем, что в данном примере мы не использовали обогащение данных, которое необходимо для предотвращения переобучения на небольших наборах изображений.

**Выделение признаков с обогащением данных**

Теперь рассмотрим второй прием выделения признаков, более медленный и затратный, но позволяющий использовать обогащение данных в процессе обучения, — объединение модели `conv_base` с новым полносвязным классификатором и ее полноценное обучение.

Для этого мы сначала заморозим сверточную основу. Замораживание одного или нескольких слоев предотвращает изменение весовых коэффициентов в них в процессе обучения. Если этого не сделать, 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`.

**Листинг 8.23.** Создание и заморозка сверточной основы

```
conv_base = keras.applications.vgg16.VGG16(  
    weights="imagenet",  
    include_top=False)  
conv_base.trainable = False
```

При передаче в атрибуте `trainable` значения `False` список обучаемых весов слоя или модели очищается.

**Листинг 8.24.** Вывод списка обучаемых весов до и после заморозки

```
>>> conv_base.trainable = True  
>>> print("This is the number of trainable weights "  
      "before freezing the conv base:", len(conv_base.trainable_weights))  
This is the number of trainable weights before freezing the conv base: 26  
>>> conv_base.trainable = False  
>>> print("This is the number of trainable weights "  
      "after freezing the conv base:", len(conv_base.trainable_weights))  
This is the number of trainable weights after freezing the conv base: 0
```

Теперь создадим новую модель, объединяющую следующее.

1. Этап обогащения данных.
2. Замороженную сверточную основу.
3. Полносвязный классификатор.

**Листинг 8.25.** Добавление этапа обогащения данных и классификатора к сверточной основе

```
data_augmentation = keras.Sequential(  
    [  
        layers.RandomFlip("horizontal"),  
        layers.RandomRotation(0.1),  
        layers.RandomZoom(0.2),  
    ]  
)  
  
inputs = keras.Input(shape=(180, 180, 3))  
x = data_augmentation(inputs) ← Обогащение  
x = keras.applications.vgg16.preprocess_input(x) ← данных  
x = conv_base(x)  
x = layers.Flatten()(x)  
x = layers.Dense(256)(x)  
x = layers.Dropout(0.5)(x)  
outputs = layers.Dense(1, activation="sigmoid")(x)  
model = keras.Model(inputs, outputs)  
model.compile(loss="binary_crossentropy",  
              optimizer="rmsprop",  
              metrics=["accuracy"])  
  
Масштабирование  
входных данных
```

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.

Давайте начнем обучение модели. Мы добавили этап обогащения данных, поэтому обучение будет длиться намного дольше, прежде чем проявится эффект переобучения, — так что можно увеличить количество эпох, скажем, до 50.

### ПРИМЕЧАНИЕ

Этот прием настолько затратный, что его следует применять только при наличии доступа к GPU (например, к бесплатному GPU в Colab) — он абсолютно не под силу CPU. Если у вас нет возможности запустить свой код на GPU, то первый путь остается для вас единственным доступным решением.

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction_with_data_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

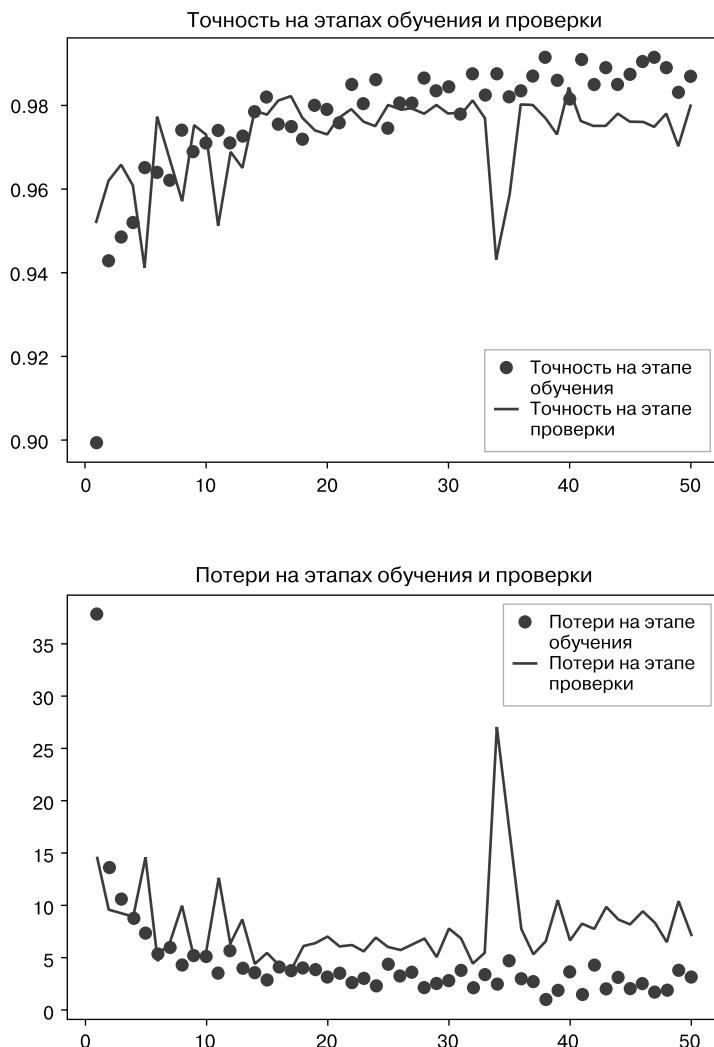
Снова построим графики изменения метрик (рис. 8.14). Как видите, мы превысили уровень точности 98 % на этапе проверки. Это серьезное улучшение по сравнению с предыдущей моделью.

Проверим точность на контрольных данных.

#### **Листинг 8.26.** Оценка модели на контрольном наборе данных

```
test_model = keras.models.load_model(
    "feature_extraction_with_data_augmentation.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Мы получили точность на контрольных данных 97,5 %. По сравнению с предыдущей моделью это довольно скромное улучшение — что немного разочаровывает, особенно учитывая, насколько улучшилась точность на проверочных данных. Точность модели всегда зависит от набора образцов, на котором выполняется ее оценка! Некоторые наборы могут содержать образцы, более трудные для распознавания, поэтому хорошие результаты на одном наборе не обязательно будут подтверждаться на всех других наборах.



**Рис. 8.14.** Изменение метрик на этапах проверки и обучения для извлечения признаков с обогащением данных

### 8.3.2. Дообучение предварительно обученной модели

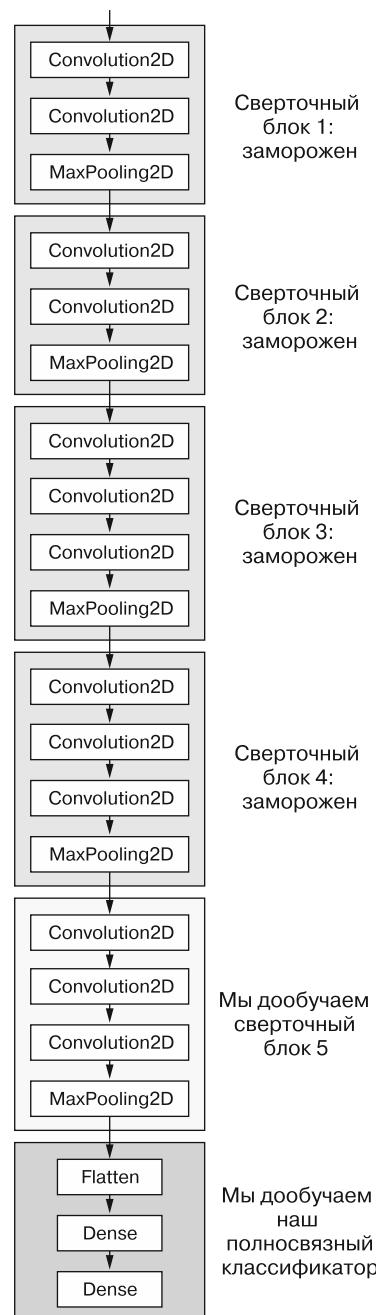
Другой широко используемый прием повторного использования модели, дополняющий выделение признаков, — *дообучение* (fine-tuning) (рис. 8.15). Дообучение заключается в размораживании нескольких верхних слоев замороженной

модели, которая использовалась для выделения признаков, и совместном обучении вновь добавленной части модели (в данном случае полносвязного классификатора) и этих верхних слоев. Данный прием называется *дообучением*, поскольку немного корректирует наиболее абстрактные представления в повторно используемой модели, чтобы сделать их более актуальными для конкретной задачи.

Выше я отмечал, что для обучения классификатора, инициализированного случайными значениями, необходимо заморозить сверточную основу сети VGG16. По той же причине дообучить несколько верхних слоев сверточной основы можно только после обучения классификатора. 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. Поэтому для дообучения сети нужно выполнить следующие шаги.

1. Добавить свою сеть поверх обученной базовой сети.
2. Заморозить базовую сеть.
3. Обучить добавленную часть.
4. Разморозить несколько слоев в базовой сети. (Обратите внимание, что не следует размноживать слои «пакетной нормализации», которые здесь неактуальны, поскольку в VGG16 таких слоев нет, — я объясню пакетную нормализацию и покажу ее влияние на дообучение в следующей главе.)
5. Обучить эти слои и добавленную часть вместе.

Мы уже выполнили первые три шага в ходе выделения признаков. Теперь выполним шаг 4: размножим `conv_base` и заморозим отдельные слои в ней.



**Рис. 8.15.** Дообучение последнего сверточного блока сети VGG16

Вспомним, как выглядит наша сверточная основа:

```
>>> conv_base.summary()  
Model: "vgg16"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_19 (InputLayer)	[None, 180, 180, 3]	0
block1_conv1 (Conv2D)	(None, 180, 180, 64)	1792
block1_conv2 (Conv2D)	(None, 180, 180, 64)	36928
block1_pool (MaxPooling2D)	(None, 90, 90, 64)	0
block2_conv1 (Conv2D)	(None, 90, 90, 128)	73856
block2_conv2 (Conv2D)	(None, 90, 90, 128)	147584
block2_pool (MaxPooling2D)	(None, 45, 45, 128)	0
block3_conv1 (Conv2D)	(None, 45, 45, 256)	295168
block3_conv2 (Conv2D)	(None, 45, 45, 256)	590080
block3_conv3 (Conv2D)	(None, 45, 45, 256)	590080
block3_pool (MaxPooling2D)	(None, 22, 22, 256)	0
block4_conv1 (Conv2D)	(None, 22, 22, 512)	1180160
block4_conv2 (Conv2D)	(None, 22, 22, 512)	2359808
block4_conv3 (Conv2D)	(None, 22, 22, 512)	2359808
block4_pool (MaxPooling2D)	(None, 11, 11, 512)	0
block5_conv1 (Conv2D)	(None, 11, 11, 512)	2359808
block5_conv2 (Conv2D)	(None, 11, 11, 512)	2359808
block5_conv3 (Conv2D)	(None, 11, 11, 512)	2359808
block5_pool (MaxPooling2D)	(None, 5, 5, 512)	0
<hr/>		
Total params: 14,714,688		
Trainable params: 14,714,688		
Non-trainable params: 0		

---

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 2–3 слоя сверточной основы. Сделаем это, начав с того места, на котором мы остановились в предыдущем примере.

#### **Листинг 8.27.** Замораживание всех слоев, кроме заданных

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
    layer.trainable = False
```

Теперь можно начинать дообучение модели. 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 8.28.** Дообучение модели

```
model.compile(loss="binary_crossentropy",
              optimizer=keras.optimizers.RMSprop(learning_rate=1e-5),
              metrics=["accuracy"])
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="fine_tuning.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

Закончив обучение, оценим модель на контрольных данных:

```
model = keras.models.load_model("fine_tuning.keras")
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Мы получили точность на уровне 98,5 % (и снова вы можете получить другую цифру, с разницей в пределах одного процента). В оригинальном состязании на сайте Kaggle, основанном на этом наборе данных, это был бы один из лучших результатов. Впрочем, сравнивать наши условия с конкурсными не совсем справедливо: мы использовали предварительно обученную модель, уже обладающую знаниями, которые помогали ей отличать кошек от собак, тогда как участники состязания такой возможности не имели.

С другой стороны, благодаря современным методам глубокого обучения нам удалось достичь такого результата, использовав лишь малую часть обучающих данных (около 10 %), доступных участникам соревнования. Между обучением на 20 000 и на 2000 образцов огромная разница!

Теперь у вас есть надежный набор инструментов для решения задач классификации изображений, особенно с ограниченным объемом данных.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Сверточные нейронные сети — лучший тип моделей машинного обучения для задач распознавания образов. Такую сеть можно обучить с нуля на очень небольшом наборе данных — и получить приличный результат.
- Сверточные нейронные сети создают иерархию модульных шаблонов и понятий для представления видимого мира.
- Когда объем данных ограничен, главной проблемой становится переобучение. Обогащение 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.

# *Продвинутые приемы глубокого обучения в технологиях компьютерного зрения*

---

## **В этой главе**

- ✓ Различные сферы компьютерного зрения: классификация изображений, сегментация изображений, обнаружение объектов.
- ✓ Современные архитектуры сверточных нейронных сетей: остаточные связи, пакетная нормализация, раздельные свертки по глубине.
- ✓ Методы визуализации и интерпретации знаний, заключенных в сверточной сети.

Ранее вы познакомились с основами использования глубокого обучения в технологиях компьютерного зрения на примере простых моделей (стеки слоев Conv2D и MaxPooling2D), применяемых для решения простой задачи (бинарная классификация изображений). Но компьютерное зрение — это не только классификация изображений! В этой главе будет рассмотрен более широкий круг задач и представлены некоторые продвинутые приемы.

## **9.1. ТРИ ОСНОВНЫЕ ЗАДАЧИ В СФЕРЕ КОМПЬЮТЕРНОГО ЗРЕНИЯ**

До сих пор наше внимание было сосредоточено на моделях классификации изображений, которые принимают изображение и возвращают соответствующую ему метку. «Это изображение, вероятно, содержит кошку; а на этом, вероятно, существует собака». Но классификация изображений — лишь один из нескольких

возможных вариантов применения глубокого обучения в компьютерном зрении. В целом в данной сфере есть три основные задачи.

- *Классификация изображений.* Цель — присвоить изображению одну или несколько меток. Это может быть однозначная классификация (изображение можно отнести только к одной категории) или многозначная (изображению можно присвоить несколько меток, в зависимости от наличия на нем объектов из разных категорий, как на рис. 9.1). Например, когда вы выполняете поиск по ключевому слову в приложении Google Photo, ваш запрос обрабатывается очень большой моделью многозначной классификации, распознающей более 20 000 различных классов и обученной на миллионах изображений.

Однозначная многоклассовая классификация



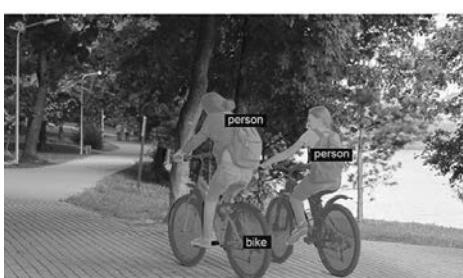
- Велопрогулка
- Бег
- Плавание

Многозначная классификация

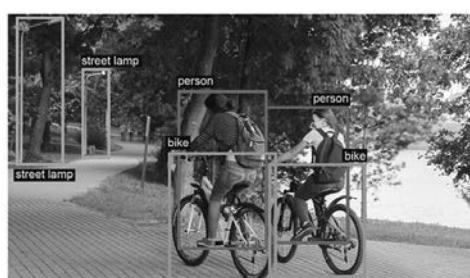


- |   |  |
|---|--|
| <input checked="" type="checkbox"/> Велосипед | <input checked="" type="checkbox"/> Дерево |
| <input checked="" type="checkbox"/> Человек   | <input type="checkbox"/> Автомобиль        |
| <input type="checkbox"/> Лодка                | <input type="checkbox"/> Дом               |

Сегментация изображений



Обнаружение объектов



**Рис. 9.1.** Три основные задачи компьютерного зрения: классификация и обнаружение объектов

- *Сегментация изображений.* Цель — сегментировать, или разбить, изображение на непересекающиеся области, каждая из которых представляет некоторую категорию (как на рис. 9.1). Например, когда в Zoom или Google Meet вам нужно установить отображение какого-то фона за собой во время видеозвонка, для реализации этой опции программа использует модель сег-

ментации изображения, отличающую ваше лицо от фона за ним с точностью до пикселя.

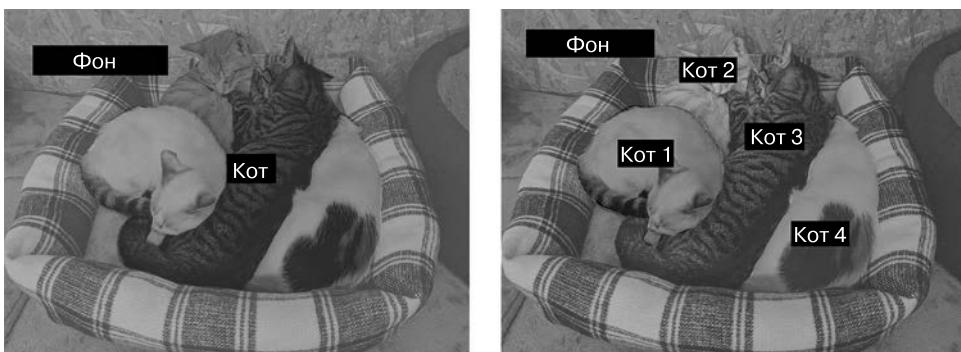
- *Обнаружение объектов.* Цель — нарисовать прямоугольники (которые называют *ограничивающими рамками*) вокруг интересующих объектов на изображении и связать каждый прямоугольник с некоторым классом. Автопилот автомобиля, например, может использовать модель обнаружения объектов для наблюдения за автомобилями, пешеходами и знаками, попадающими в поле зрения своих камер.

## 9.2. ПРИМЕР СЕГМЕНТАЦИИ ИЗОБРАЖЕНИЯ

Чтобы сегментировать изображение, в глубоком обучении используется модель для назначения класса каждому пикселю изображения — таким образом изображение *сегментируется* на разные зоны (например «фон» и «передний план» или «дорога», «автомобиль» и «тротуар»). Данная категория методов может найти применение в инструментах редактирования изображений и видео, автоматическом управлении транспортными средствами, робототехнике, медицине и т. д.

Есть два разных варианта сегментации изображений (рис. 9.2).

- *Семантическая сегментация* — когда каждый пиксель независимо друг от друга относится к некоторой семантической категории (скажем, «кошка»). В этом случае, если на изображении есть две кошки, все соответствующие пиксели будут включены в одну и ту же общую категорию — «кошка» (рис. 9.2).
- *Сегментация экземпляров* — направлена не только на классификацию пикселей изображения по категориям, но и на выделение отдельных экземпляров объекта. Поэтому на изображении с двумя кошками экземпляры «кошка 1» и «кошка 2» будут распознаны как две отдельные группы пикселей.



**Рис. 9.2.** Семантическая сегментация и сегментация экземпляров

Сосредоточимся на семантической сегментации: в примере ниже мы еще раз исследуем изображения кошек и собак, но на этот раз постараемся научиться различать основной предмет и его фон.

Для работы нам понадобится набор данных Oxford-IIIT Pets ([www.robots.ox.ac.uk/~vgg/data/pets/](http://www.robots.ox.ac.uk/~vgg/data/pets/)), содержащий 7390 изображений различных пород кошек и собак вместе с соответствующими масками сегментации. *Маска сегментации* — это эквивалент метки в задаче сегментации изображения; изображение того же размера, что и входное изображение, с одним цветовым каналом, в котором каждое целочисленное значение обозначает класс соответствующего пикселя на входном изображении. В нашем случае пиксели масок сегментации могут принимать одно из трех целочисленных значений:

- 1 (передний план);
- 2 (фон);
- 3 (контур).

Для начала загрузим и распакуем набор данных, используя утилиты `wget` и `tar`:

```
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
!tar -xf images.tar.gz
!tar -xf annotations.tar.gz
```

Входные изображения сохраняются в виде файлов JPEG в папке `images/` (например, `images/Abyssinian_1.jpg`), а соответствующие маски сегментации — в виде файлов PNG с теми же именами в папке `annotations/trimaps/` (например, `annotations/trimaps/Abyssinian_1.png`).

Подготовим список путей к входным файлам и файлам масок:

```
import os

input_dir = "images/"
target_dir = "annotations/trimaps/"

input_img_paths = sorted(
    [os.path.join(input_dir, fname)
     for fname in os.listdir(input_dir)
     if fname.endswith(".jpg")])
target_paths = sorted(
    [os.path.join(target_dir, fname)
     for fname in os.listdir(target_dir)
     if fname.endswith(".png") and not fname.startswith(".")])
```

А теперь посмотрим, как выглядит одно из входных изображений и его маска. Выведем на экран изображение (рис. 9.3):

```
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array

plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

← Вывести девятое  
входное изображение



**Рис. 9.3.** Пример изображения

и его маску (рис. 9.4):



**Рис. 9.4.** Соответствующая целевая маска

```

def display_target(target_array):
    normalized_array = (target_array.astype("uint8") - 1) * 127 ←
    plt.axis("off")
    plt.imshow(normalized_array[:, :, 0])

img = img_to_array(load_img(target_paths[9], color_mode="grayscale")) ←
display_target(img)

```

В исходном наборе данных метки имеют значения 1, 2 и 3. Мы вычитаем 1, чтобы привести метки в диапазон от 0 до 2, а затем умножаем на 127, чтобы получить метки 0 (черный цвет), 127 (серый), 254 (почти белый)

Аргумент `color_mode="grayscale"` обеспечивает обработку загружаемого изображения как имеющего единственный цветовой канал

Далее загрузим входные данные и цели в два массива NumPy и разделим массивы на обучающий и проверочный. Так как набор данных невелик, его можно целиком загрузить в память:

```

import numpy as np
import random

img_size = (200, 200) ← Все изображения будут масштабироваться до размеров 200 × 200
num_imgs = len(input_img_paths) ← Общее количество образцов

random.Random(1337).shuffle(input_img_paths) | ← Перемешивание файлов (изначально они были отсортированы по породам).
random.Random(1337).shuffle(target_paths) | ← В обеих инструкциях используется одно и то же начальное число (1337), чтобы гарантировать единый порядок входных и целевых файлов

def path_to_input_image(path):
    return img_to_array(load_img(path, target_size=img_size))

def path_to_target(path):
    img = img_to_array(
        load_img(path, target_size=img_size, color_mode="grayscale"))
    img = img.astype("uint8") - 1 ← Вычитается 1, чтобы преобразовать метки в 0, 1 и 2
    return img

input_imgs = np.zeros((num_imgs,) + img_size + (3,), dtype="float32")
targets = np.zeros((num_imgs,) + img_size + (1,), dtype="uint8")
for i in range(num_imgs):
    input_imgs[i] = path_to_input_image(input_img_paths[i])
    targets[i] = path_to_target(target_paths[i])

num_val_samples = 1000
train_input_imgs = input_imgs[:-num_val_samples]
train_targets = targets[:-num_val_samples]
val_input_imgs = input_imgs[-num_val_samples:]
val_targets = targets[-num_val_samples:]

Разделение данных на обучающий и проверочный наборы Резервирование 1000 образцов для проверочного набора

```

Загрузка всех изображений в массив `input_imgs` типа `float32` и масок в массив `targets` типа `uint8` (в одном и том же порядке). Входные изображения имеют три канала (значения RGB), а цели — один канал (с целочисленными метками)

Теперь определим модель:

```

from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs) ←

    Не забывайте
    изменять масштаб
    входных значений
    до диапазона [0-1]

    ←
    Обратите внимание,
    что мы везде используем
    padding="same", чтобы
    избежать влияния отступов
    от границ на размер
    карты объектов

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x) ←
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
                           padding="same")(x) ←

    Модель завершается классификатором
    пикселей с активацией softmax,
    который относит каждый выходной
    пиксель к одной из трех категорий

    ←
    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()

```

Вот вывод метода `model.summary()`:

Model: "model"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None, 200, 200, 3)]	0
rescaling (Rescaling)	(None, 200, 200, 3)	0
conv2d (Conv2D)	(None, 100, 100, 64)	1792
conv2d_1 (Conv2D)	(None, 100, 100, 64)	36928
conv2d_2 (Conv2D)	(None, 50, 50, 128)	73856
conv2d_3 (Conv2D)	(None, 50, 50, 128)	147584

conv2d_4 (Conv2D)	(None, 25, 25, 256)	295168
conv2d_5 (Conv2D)	(None, 25, 25, 256)	590080
conv2d_transpose (Conv2DTran)	(None, 25, 25, 256)	590080
conv2d_transpose_1 (Conv2DTr)	(None, 50, 50, 256)	590080
conv2d_transpose_2 (Conv2DTr)	(None, 50, 50, 128)	295040
conv2d_transpose_3 (Conv2DTr)	(None, 100, 100, 128)	147584
conv2d_transpose_4 (Conv2DTr)	(None, 100, 100, 64)	73792
conv2d_transpose_5 (Conv2DTr)	(None, 200, 200, 64)	36928
conv2d_6 (Conv2D)	(None, 200, 200, 3)	1731
<hr/>		
Total params:	2,880,643	
Trainable params:	2,880,643	
Non-trainable params:	0	

---

Первая половина модели очень похожа на типичную сверточную сеть, используемую для классификации изображений: стек слоев Conv2D с постепенно увеличивающимися размерами фильтров. Мы трижды повторяем уменьшение наполовину разрешения наших изображений, что дает активацию с размерами (25, 25, 256). Цель этой первой половины – закодировать изображения в карты признаков меньшего размера, где каждое пространственное местоположение (или пиксель) содержит информацию о большем пространственном фрагменте исходного изображения. Конечный эффект можно интерпретировать как сжатие.

Одно из важных различий между первой половиной этой модели и моделями классификации, которые были показаны ранее в этой книге, заключается в способе уменьшения разрешения. В сверточных сетях в предыдущей главе для уменьшения разрешения карт признаков мы использовали слои MaxPooling2D. Здесь мы увеличили *шаг свертки* во всех остальных сверточных слоях (если вы забыли, какой эффект оказывает изменение шага свертки, вернитесь к подразделу «Шаг свертки» в пункте 8.1.1). В случае сегментации изображения важно позаботиться о *пространственной привязке* информации на изображении, так как на выходе модели должны создаваться попиксельные целевые маски. Поэтому мы выбрали данное решение. Применение метода на основе выбора максимального значения из соседних в окне  $2 \times 2$  полностью уничтожит информацию о местоположении в каждом таком окне: вы получите одно скалярное значение для каждого окна, не имея представления, из которого из четырех местоположений в окне оно было получено. То есть, несмотря на то что данный метод хорошо подходит для задач классификации, в задаче сегментации он нам только навредит. Между тем

свертки с увеличенным шагом лучше справляются с уменьшением разрешения карт признаков, когда требуется сохранить информацию о местоположении. Как вы не раз увидите далее в этой книге, мы будем использовать именно прием увеличения шага свертки вместо выбора максимального значения из соседних в любых моделях, где важно сохранить информацию о местоположении признаков — например, в генеративных моделях в главе 12.

Вторая половина модели организована как стек слоев `Conv2DTranspose`. Что это за слои? Дело в том, что результатом первой половины модели является карта признаков с формой  $(25, 25, 256)$ , а нам нужно получить на выходе результат той же формы, что и целевые маски  $(200, 200, 3)$ . То есть к выходным данным первой половины модели нужно применить преобразования, увеличивающие разрешение карт признаков. Именно такое преобразование реализует слой `Conv2DTranspose`: его можно рассматривать как сверточный слой, который *учится увеличивать разрешение*. Если исходные данные с формой  $(100, 100, 64)$  пропустить через слой `Conv2D(128, 3, strides=2, padding="same")`, то на выходе получится результат формы  $(50, 50, 128)$ . Если этот вывод пропустить через слой `Conv2DTranspose(64, 3, strides=2, padding="same")`, получим результат с формой  $(100, 100, 64)$ , как и у оригинала. То есть после сжатия входных данных в карты признаков с формой  $(25, 25, 256)$  с помощью стека слоев `Conv2D` можно просто применить соответствующую последовательность слоев `Conv2DTranspose`, чтобы получить изображения с формой  $(200, 200, 3)$ .

Теперь скомпилируем и обучим модель:

```
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy")

callbacks = [
    keras.callbacks.ModelCheckpoint("oxford_segmentation.keras",
                                    save_best_only=True)
]

history = model.fit(train_input_imgs, train_targets,
                     epochs=50,
                     callbacks=callbacks,
                     batch_size=64,
                     validation_data=(val_input_imgs, val_targets))
```

и построим графики изменения потерь на этапах обучения и проверки (рис. 9.5):

```
epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
val_loss = history.history["val_loss"]
plt.figure()
plt.plot(epochs, loss, "bo", label="Потери на этапе обучения")
plt.plot(epochs, val_loss, "b", label="Потери на этапе проверки")
plt.title("Потери на этапах обучения и проверки")
plt.legend()
```

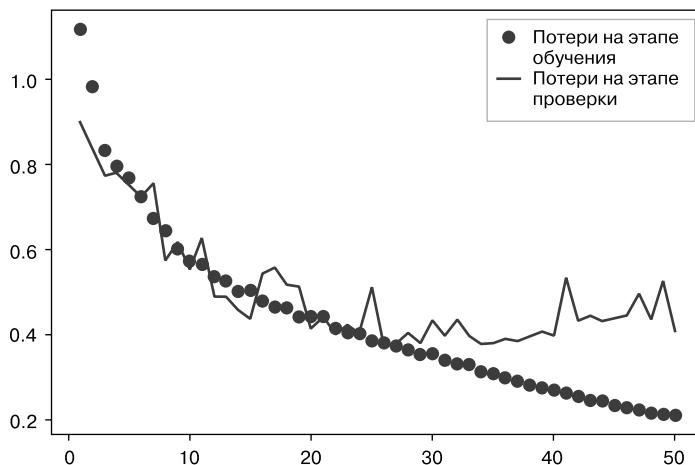


Рис. 9.5. Графики изменения потерь на этапах обучения и проверки

Как видите, на полпути (где-то в районе 25-й эпохи) начал проявляться эффект переобучения. Загрузим модель, показавшую наименьшие потери на проверочных данных, и посмотрим, как ее использовать для прогнозирования маски сегментации (рис. 9.6):

```
from tensorflow.keras.utils import array_to_img
model = keras.models.load_model("oxford_segmentation.keras")
i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))

mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

В получившейся маске есть пара небольших артефактов, обусловленных присутствием геометрических фигур на переднем и заднем планах, но в целом модель дала довольно точный результат.

До сих пор (на протяжении всей главы 8 и в начале главы 9) вы знакомились с основами классификации и сегментации изображений. С имеющимися теперь у вас знаниями можно многое добиться. Тем не менее сверточные нейронные сети, которые опытные инженеры разрабатывают для решения реальных задач, не так просты, как в наших примерах. Вы пока не сможете, подобно экспертам, в процессе конструирования моделей быстро принимать точные решения. Чтобы восполнить этот пробел, вам нужно познакомиться с архитектурными шаблонами. Давайте займемся этим прямо сейчас.



**Рис. 9.6.** Контрольное изображение и спрогнозированная для него маска

## 9.3. СОВРЕМЕННЫЕ АРХИТЕКТУРНЫЕ ШАБЛОНЫ СВЕРТОЧНЫХ СЕТЕЙ

«Архитектура» модели — это сумма решений, которые применялись при ее создании: использованные слои, их настройки и порядок соединения. Эти решения определяют *пространство гипотез* модели: пространство возможных функций, параметризованных весами модели, по которым градиентный спуск может выполнять поиск. Так же как при проектировании признаков, хорошее пространство гипотез кодирует *имеющиеся знания* о задаче и ее решении. Например, использование сверточных слоев предполагает предварительное знание, что соответствующие шаблоны, присутствующие в исходных изображениях, инвариантны в отношении переноса. Для эффективного обучения на данных обязательно нужно делать предположения о том, что вы ищете.

От архитектуры часто зависит успех или неудача модели. При выборе неправильной архитектуры модель может не добиться высоких показателей, и никакие обучающие данные не спасут ее. И наоборот, хорошая архитектура может ускорить обучение модели и позволит ей эффективно использовать доступные обучающие данные, уменьшая потребность в больших наборах данных. Хорошая архитектура модели *уменьшает размер области поиска*, или, иными словами, *упрощает схождение к оптимальной точке области поиска*. По аналогии с проектированием признаков и курированием данных цель архитектуры модели — *упростить задачу* для градиентного спуска. Помните, что градиентный спуск — довольно глупый поисковый процесс, поэтому ему нужна любая возможная помощь.

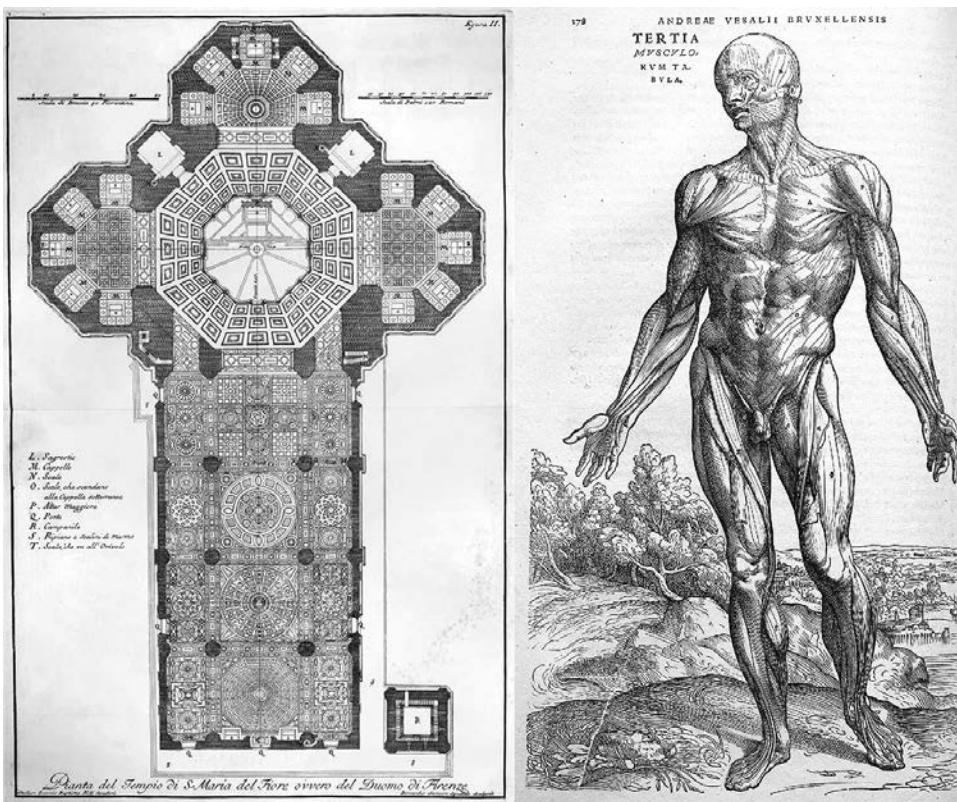
Выбор архитектуры модели — это больше искусство, чем наука. Опытные специалисты могут создавать высококачественные модели с первой попытки, основываясь исключительно на своей интуиции, тогда как новички часто испытывают сложности с разработкой модели, способной к обучению. Ключевое слово здесь — *интуиция*: никто не сможет четко обосновать, почему та или иная архитектура годится или не годится. Эксперты полагаются на опыт, приобретаемый в процессе решения разнообразных практических задач. В процессе чтения этого издания вы разовьете собственную интуицию. Однако дело *не только* в ней — хоть в науке глубокого обучения мало чего-то особенного, но, как в любой инженерной дисциплине, есть свои лучшие практики.

В следующих разделах мы рассмотрим несколько таких практик для архитектуры сверточных сетей, в частности *остаточные связи*, *пакетную нормализацию* и *раздельные свертки*. С их помощью вы сможете создавать высокоэффективные модели распознавания изображений. Мы применим эти практики в нашей задаче классификации изображений кошек и собак.

Начнем с общей организации архитектуры по формуле «модульность — иерархия — многократное использование» (modularity-hierarchy-reuse, MHR).

### **9.3.1. Модульность, иерархия, многократное использование**

Есть универсальный рецепт, помогающий упростить сложную систему: нужно лишь структурировать всю аморфную мешанину в *модули*, организовать модули в *иерархию* и *многократно использовать* одни и те же модули в разных местах по мере необходимости («многократное использование» — еще один термин для обозначения *абстракции* в этом контексте). Формула «модульность — иерархия — многократное использование» лежит в основе системной архитектуры практически во всех областях, где в принципе используется термин «архитектура». На ней базируется любая сложная система, будь то собор, ваше собственное тело, военно-морской флот страны или кодовая база Keras (рис. 9.7).



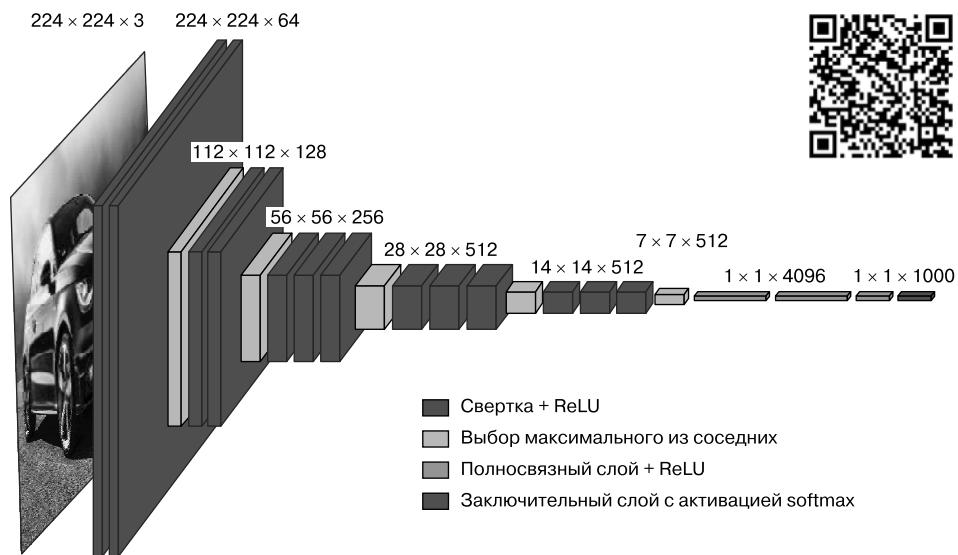
**Рис. 9.7.** Сложные системы организованы в модули с иерархической структурой, которые многократно используются (например, ваши четыре конечности или ваши 20 пальцев являются вариантами одного и того же шаблона)

Инженеры-программисты хорошо знакомы с этими принципами: эффективная кодовая база имеет иерархическую модульную организацию, в которой одни и те же классы и функции используются повторно, а не реализуются дважды. Если вы организуете свой код, следуя этим принципам, то можно сказать, что вы планируете «архитектуру программного обеспечения».

Само по себе глубокое обучение — всего лишь применение данного рецепта к непрерывной оптимизации методом градиентного спуска: вы берете классический метод оптимизации (градиентный спуск по непрерывному функциональному пространству) и структурируете пространство поиска на модули (слои), организованные в иерархию (часто в виде простого стека — простейшей разновидности иерархии), в которой повторно используется все что можно (например, свертки предназначены для повторного применения одной и той же информации в разных пространственных местоположениях).

Точно так же архитектура модели глубокого обучения в первую очередь связана с разумным применением принципов модульности, иерархии и повторного использования. В своей практике вы будете замечать, что все популярные сверточные архитектуры не только структурированы по слоям — их слои также организованы в повторяющиеся группы, называемые блоками или модулями. Например, популярная архитектура VGG16, которую мы использовали в предыдущей главе, имеет повторяющиеся блоки «свертка, свертка, выбор максимального» (рис. 9.8).

Кроме того, большинство сверточных сетей имеют пирамидальную структуру (*иерархию признаков*). Вспомните, например, последовательное увеличение количества сверточных фильтров, которые мы использовали в первой сверточной сети, построенной в предыдущей главе: 32, 64, 128. Число фильтров растет с глубиной слоя, а размер карт признаков, соответственно, уменьшается. То же самое можно заметить в организации модели VGG16 (рис. 9.8).



**Рис. 9.8.** Архитектура VGG16: обратите внимание на повторяющиеся блоки слоев и пирамидальную структуру карт признаков

Более глубокие иерархии хороши тем, что способствуют повторному использованию признаков и, следовательно, абстрагированию. В общем случае глубокий стек слоев меньшей размерности работает лучше, чем мелкий стек слоев большой размерности. Однако количество слоев не может увеличиваться до бесконечности из-за проблемы *затухания градиента*. Для ее преодоления используется важный архитектурный шаблон — остаточные связи.

### О ВАЖНОСТИ ИССЛЕДОВАНИЯ ВОЗМОЖНОСТИ УПРОЩЕНИЯ МОДЕЛЕЙ ГЛУБОКОГО ОБУЧЕНИЯ

Архитектуры моделей глубокого обучения чаще являются результатом постепенного развития, чем проектирования, — они разрабатываются методом проб и ошибок. Так же как в биологических системах, если взять любую сложную экспериментальную модель глубокого обучения, есть вероятность, что удаление нескольких модулей (или замена некоторых обученных весов случайными значениями) не приведет к потере качества.

Наращивая сложность, исследователи глубокого обучения сталкиваются со следующим соблазном: сложная (даже более, чем необходимо) система может быть более интересной или более новой — что увеличит шансы прохождения авторских статей о ней через процесс рецензирования. Прочитав множество трудов по глубокому обучению, вы заметите, что они часто подгоняются под требования рецензентов как по стилю, так и по содержанию — и это отрицательно сказывается на ясности объяснения и надежности результатов. Например, математический аппарат в подобных текстах редко используется для четкой формализации идей или неочевидных результатов — чаще он применяется для придания налета серьезности, как дорогой костюм продавца.

Целью исследований должна быть не только публикация, но и получение надежных знаний. Самый простой путь для этого — изучение *причинно-следственной связи* в вашей системе. Есть очень простой способ изучения причинно-следственной связи: *исследование аблации* (возможности упрощения модели). Оно предполагает систематические попытки упростить систему удалением ее частей, чтобы определить, где в действительности формируются основные результаты. Если вы обнаружите, что  $X + Y + Z$  дает хорошие результаты, попробуйте также  $X$ ,  $Y$ ,  $Z$ ,  $X + Y$ ,  $X + Z$  и  $Y + Z$  и посмотрите, что будет.

Став исследователем глубокого обучения, старайтесь избавляться от шума в процессе исследований: изучайте возможность упрощения своих моделей. Всегда спрашивайте: «Есть ли более простое объяснение? Действительно ли необходима эта дополнительная сложность? Зачем?»

#### 9.3.2. Остаточные связи

Возможно, вы знакомы с детской игрой «испорченный телефон» (в Великобритании ее называют также «китайским шепотом», а во Франции — «арабским телефоном»), где первоначальное сообщение нашептывается на ухо первому игроку, который затем нашептывает его на ухо следующему и т. д. Последний игрок громко сообщает услышанное им сообщение, зачастую существенно

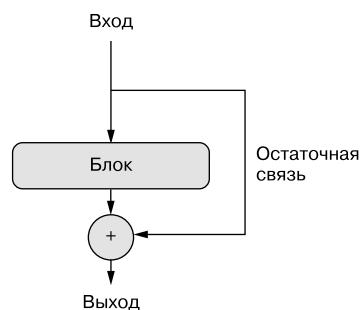
отличающееся от исходной версии. Это — забавная метафора накопления ошибок при последовательной передаче информации по зашумленному каналу.

Как оказалось, обратное распространение в последовательной модели глубокого обучения очень похоже на игру в «испорченный телефон». У вас есть цепочка функций, например:

$$y = f_4(f_3(f_2(f_1(x))))$$

Цель игры — настроить параметры каждой функции в цепочке, основываясь на ошибке, полученной на выходе  $f_4$  (потеря модели). Чтобы настроить  $f_1$ , нужно передать информацию об ошибке через  $f_2$ ,  $f_3$  и  $f_4$ . Однако каждая следующая функция в цепочке вносит свои искажения. Если цепочка функций слишком глубокая, искажения начинают подавлять информацию о градиенте, и обратное распространение перестает работать. Ваша модель вообще не будет обучаться. Это проблема *затухания градиентов*.

Решение простое: нужно лишь заставить каждую функцию в цепочке перестать вносить искажения, чтобы сохранить информацию, полученную от предыдущей функции. Самый простой способ реализовать это — использовать *остаточные связи*. Входные данные слоя или блока слоев добавляются в его выходные данные (рис. 9.9). Остаточные связи действуют как *короткие пути* для распространения информации в обход деструктивных блоков или блоков, вносящих существенные искажения (таких как блоки с нежелательными активациями или слоями прореживания), позволяя информации градиента ошибок проходить по глубокой сети без искажений. Этот метод был представлен в 2015 году в семействе моделей ResNet (разработанном Каймином Хе с коллегами в Microsoft)<sup>1</sup>.



**Рис. 9.9.** Остаточная связь в обход блока, выполняющего обработку

На практике остаточная связь реализуется следующим образом.

#### Листинг 9.1. Реализация остаточной связи в псевдокоде

```

x = ...           Некоторый входной тензор
residual = x    Сохранить указатель на исходные данные
x = block(x)    Это вычислительный блок, который может вносить искажения
x = add([x, residual]) Добавить исходные данные в выход слоя: получившиеся выходные
                       данные будут содержать полную информацию о входе
  
```

<sup>1</sup> He Kaiming et al. Deep Residual Learning for Image Recognition // Conference on Computer Vision and Pattern Recognition, 2015, <https://arxiv.org/abs/1512.03385>.

Обратите внимание: добавление входных данных блока в выходные подразумевает, что выход должен иметь ту же форму, что и вход. Однако этот прием не подходит для случаев, когда блок включает сверточные слои с увеличенным количеством фильтров или слой выбора максимального по соседям. В таких случаях можно использовать слой Conv2D  $1 \times 1$  без активации для линейного проецирования остатков в желаемую форму вывода (листинг 9.2). Обычно сверточные слои в целевом блоке создаются с аргументом padding="same", чтобы избежать уменьшения пространственного разрешения из-за дополнения, и берутся увеличенные шаги свертки в остаточной проекции, чтобы обеспечить соответствие любому уменьшению пространственного разрешения, вызванному слоем выбора максимального по соседним значениям (листинг 9.3).

### Листинг 9.2. Остаточный блок, в котором изменяется число фильтров

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x) ← Сохранить исходные данные
residual = layers.Conv2D(64, 1)(residual) ← для остаточной связи
x = layers.add([x, residual]) ← Это слой, в обход которого создается
                                остаточная связь: он увеличивает
                                количество фильтров на выходе
                                с 32 до 64. Обратите внимание, что
                                аргумент padding="same" используется
                                здесь для того, чтобы избежать
                                уменьшения разрешения из-за дополнения

Теперь выход блока и тензор residual имеют
одинаковую форму, и их можно сложить

В residual имеется только 32 фильтра, поэтому мы используем
слой Conv2D  $1 \times 1$  для преобразования в требуемую форму
```

### Листинг 9.3. Случай, когда целевой блок включает слой выбора максимального по соседям

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x) ← Сохранить исходные данные
x = layers.MaxPooling2D(2, padding="same")(x) ← для остаточной связи
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual]) ← Блок из двух слоев, вокруг которого создается
                                остаточная связь: он включает слой  $2 \times 2$ 
                                выбора максимального по соседям. Обратите
                                внимание, что аргумент padding="same"
                                используется здесь в обоих слоях — Conv2D
                                и MaxPooling2D, — чтобы избежать
                                уменьшения разрешения из-за дополнения

Теперь выход блока
и тензор residual имеют
одинаковую форму,
и их можно сложить

В слое преобразования остатков
используется аргумент strides=2,
чтобы обеспечить соответствие
с уменьшенным разрешением,
созданным слоем MaxPooling2D
```

Для большей конкретности ниже приводится пример простой сверточной сети, организованной в серию блоков, каждый из которых состоит из двух сверточных слоев и одного необязательного слоя MaxPooling2D с остаточными связями в обход каждого блока:

```

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
model.summary()

```

Первый блок

Вспомогательная функция  
для создания блока с двумя  
сверточными слоями и одним  
необязательным слоем MaxPooling2D  
и остаточной связи вокруг него

Если требуется добавить  
слой MaxPooling2D,  
нужно также добавить  
сверточный слой  
с увеличенным  
шагом свертки, чтобы  
преобразовать остатки  
в необходимую форму

Последний блок создается без слоя MaxPooling2D,  
потому что далее применяется слой глобального усреднения

Второй блок; обратите внимание, что число  
фильтров в каждом блоке увеличивается

Если слой MaxPooling2D  
добавлять не требуется,  
преобразовывать  
остатки нужно, только  
если количество  
каналов изменилось

Ниже приводится сводная информация о созданной модели:

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	[(None, 32, 32, 3)]	0	
rescaling (Rescaling)	(None, 32, 32, 3)	0	input_1[0][0]
conv2d (Conv2D)	(None, 32, 32, 32)	896	rescaling[0][0]
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 16, 16, 32)	128	rescaling[0][0]
add (Add)	(None, 16, 16, 32)	0	max_pooling2d[0][0] conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 16, 16, 64)	18496	add[0][0]

conv2d_4 (Conv2D)	(None, 16, 16, 64)	36928	conv2d_3[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0	conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 8, 8, 64)	2112	add[0][0]
add_1 (Add)	(None, 8, 8, 64)	0	max_pooling2d_1[0][0] conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 8, 8, 128)	73856	add_1[0][0]
conv2d_7 (Conv2D)	(None, 8, 8, 128)	147584	conv2d_6[0][0]
conv2d_8 (Conv2D)	(None, 8, 8, 128)	8320	add_1[0][0]
add_2 (Add)	(None, 8, 8, 128)	0	conv2d_7[0][0] conv2d_8[0][0]
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0	add_2[0][0]
dense (Dense)	(None, 1)	129	global_average_pooling2d[0][0]
<hr/>			
Total params:	297,697		
Trainable params:	297,697		
Non-trainable params:	0		

С остаточными связями можно конструировать сети произвольной глубины, не беспокоясь о затухании градиентов.

Теперь перейдем к следующему важному шаблону архитектуры сверточных сетей — *пакетной нормализации*.

### 9.3.3. 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 приданье 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 и Кристианом Сегеди<sup>1</sup>; он может адаптивно нормализовать данные, даже если среднее и дисперсия изменяются во время обучения. В процессе обучения образцы нормализуются с использованием среднего и дисперсии текущего пакета данных, а во время прогнозирования (когда достаточно большой пакет репрезентативных данных может быть недоступен) применяются экспоненциальное скользящее среднее и дисперсия по всем пакетам, наблюдавшимся при обучении.

В оригинальной статье авторы утверждают, что пакетная нормализация работает за счет «уменьшения внутреннего ковариантного сдвига», но в действительности никто точно не знает, почему она способствует улучшению эффективности обучения. Есть разные гипотезы, но нет уверенности. Далее вы не раз убедитесь, что подобное положение дел характерно для многих вопросов глубокого обучения. Глубокое обучение — это не точная наука, а набор постоянно меняющихся инженерных практик, полученных опытным путем и сплетенных в единое целое ненадежными стереотипами. Иногда может казаться, что книга, которую вы держите в руках, говорит вам, *как* делать то или это, но не дает конкретного объяснения, *почему* это работает. Причина проста: мы сами этого не знаем. При наличии надежного объяснения я обязательно его упоминаю. Пакетная нормализация не относится к таким случаям.

Эффект пакетной нормализации, по всей видимости, способствует распространению градиента — подобно остаточным связям — и, соответственно, делает возможным создание более глубоких сетей. Некоторые глубокие сети могут обучаться, только если включают в себя несколько слоев `BatchNormalization`. Например, слои пакетной нормализации широко используются во многих продвинутых архитектурах сверточных нейронных сетей, входящих в состав Keras (таких как `ResNet50`, `EfficientNet` и `Xception`).

Слой `BatchNormalization` можно использовать после любого слоя — `Dense`, `Conv2D` и т. д.:

```
x = ...
x = layers.Conv2D(32, 3, use_bias=False)(x) ← | Поскольку выход слоя Conv2D
x = layers.BatchNormalization()(x)           | нормализуется, слой не нуждается
                                                | в собственном векторе смещения
```

<sup>1</sup> Ioffe Sergey and Szegedy Christian, 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>.

**ПРИМЕЧАНИЕ**

Слои обоих типов, Dense и Conv2D, включают вектор смещения (bias vector) — обучаемую переменную, цель которой — сделать слой аффинным, а не чисто линейным. Например, в общем случае слой Conv2D возвращает  $y = \text{conv}(x, \text{kernel}) + \text{bias}$ , а слой Dense —  $y = \text{dot}(x, \text{kernel}) + \text{bias}$ . Так как на этапе нормализации происходит центрирование результатов слоя по нулю, то при использовании BatchNormalization необходимость в векторе смещения отпадает и слой можно создать без него, передав параметр `use_bias=False`. Это делает слой немного тоньше.

Обычно я рекомендую размещать активацию предыдущего слоя *после* слоя пакетной нормализации (хотя это и спорно). Поэтому вместо приема, показанного в листинге 9.4, желательно использовать подход из листинга 9.5.

**Листинг 9.4.** Как не следует использовать пакетную нормализацию

```
x = layers.Conv2D(32, 3, activation="relu")(x)
x = layers.BatchNormalization()(x)
```

**Листинг 9.5.** Как следует использовать пакетную нормализацию: активация применяется после нормализации

```
x = layers.Conv2D(32, 3, use_bias=False)(x) ←
x = layers.BatchNormalization()(x) ←
x = layers.Activation("relu")(x) ←
```

Интуитивная подоплека такого выбора заключается в том, что при пакетной нормализации входные данные будут центрированы по нулю, при этом активация `relu` использует ноль как точку, где принимается решение о сохранении или отбрасывании активированных каналов, — и применение нормализации перед активацией максимизирует эффект `relu`. Однако такой способ упорядочения не особенно критичен, то есть, если выполнить свертку, затем активацию, а потом пакетную нормализацию, модель все равно будет обучаться и не обязательно покажет худшие результаты.

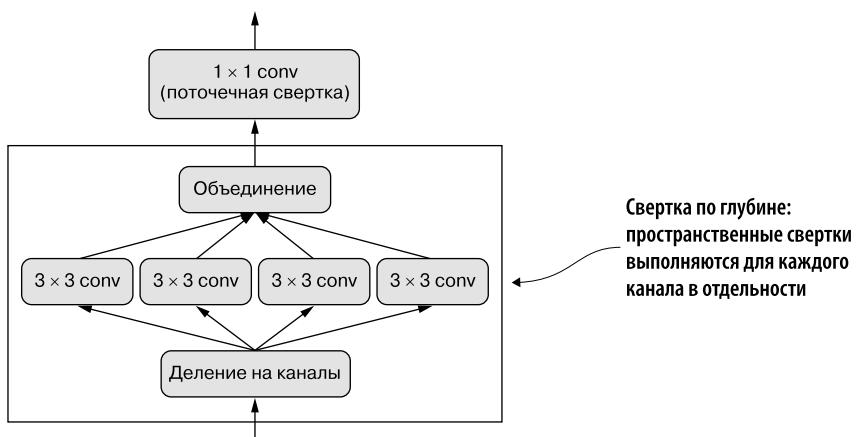
**О ПАКЕТНОЙ НОРМАЛИЗАЦИИ И ДООБУЧЕНИИ**

Применение пакетной нормализации имеет множество особенностей. Одна из них связана с дообучением: при дообучении модели, включающей слои `BatchNormalization`, я рекомендую замораживать эти слои (передавать им в атрибуте `trainable` значение `False`). Иначе они продолжат обновлять свое внутреннее среднее значение и дисперсию, что может помешать очень небольшим корректировкам, применяемым к окружающим слоям `Conv2D`.

Перейдем к следующему архитектурному шаблону: раздельной свертке по глубине.

### 9.3.4. Раздельная свертка по глубине

Что бы вы подумали, если бы я сказал, что существует такой слой, который можно использовать взамен Conv2D и таким образом 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 свойствами обладает слой *раздельной свертки по глубине* (SeparableConv2D в Keras). Этот слой выполняет пространственную свертку каждого канала во входных данных в отдельности перед смешиванием выходных каналов посредством поточечной свертки ( $1 \times 1$  conv), как показано на рис. 9.10.



**Рис. 9.10.** Раздельная свертка по глубине: за сверткой по глубине следует поточечная свертка

Это эквивалентно раздельному выделению пространственных и канальных признаков. Подобно тому как свертка основана на предположении об отсутствии связи закономерностей в изображениях с определенными местоположениями, раздельная свертка по глубине основывается на предположении *сильной корреляции пространственных местоположений* в промежуточных активациях и *практически полной независимости разных каналов*. Поскольку это предположение обычно верно для изучаемых глубокими нейронными сетями изображений, оно служит полезным предварительным условием, которое помогает модели эффективнее использовать обучающие данные. Модель с более строгими предварительными предположениями о структуре информации, которую она должна обрабатывать, является лучшей, если предварительные предположения точны.

Раздельная свертка по глубине требует намного меньше параметров и выполняет меньше вычислений по сравнению с обычной сверткой, обладая при этом сопоставимой репрезентативной мощностью. В результате получаются модели меньшего размера, которые сходятся быстрее и менее подвержены переобучению. Эти преимущества особенно важны при обучении небольших моделей с нуля на ограниченном наборе данных.

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>.

### ОДНОВРЕМЕННАЯ ЭВОЛЮЦИЯ ОБОРУДОВАНИЯ, ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И АЛГОРИТМОВ

Рассмотрим обычную операцию свертки с окном  $3 \times 3$ , 64 входными каналами и 64 выходными каналами. В ней используется  $3 \times 3 \times 64 \times 64 = 36\,864$  обучаемых параметра. При применении ее к изображению будет выполнено множество действий с вещественными числами, пропорционально количеству параметров. А теперь представьте эквивалентную раздельную свертку по глубине: она включает всего  $3 \times 3 \times 64 + 64 \times 64 = 4672$  обучаемых параметра и выполняет намного меньше действий с вещественными числами. Разница в эффективности увеличивается еще больше с увеличением количества фильтров или размеров окон свертки.

То есть при использовании раздельной свертки по глубине можно ожидать значительного ускорения? Не торопитесь с выводами. Утверждение было бы верным, если бы вы писали простые реализации алгоритмов на CUDA или C. На самом деле значительное ускорение можно увидеть на CPU при использовании распараллеленной базовой реализации на C. Но на практике вы, вероятно, берете GPU — и ваши фактические реализации весьма далеки от «простых» реализаций CUDA: это ядро cuDNN, фрагмент кода, чрезвычайно оптимизированный, вплоть до каждой машинной инструкции. Безусловно, на оптимизацию подобного кода нужно потратить много усилий, ведь свертки cuDNN на оборудовании NVIDIA выполняют много квинтиллионов операций с плавающей точкой каждый день. Однако подобная экстремальная оптимизация имеет один побочный эффект: альтернативные подходы почти не дают преимуществ в производительности, даже те с существенными внутренними преимуществами (как раздельные свертки по глубине).

<sup>1</sup> Chollet F. Xception: Deep Learning with Depthwise Separable Convolutions // Conference on Computer Vision and Pattern Recognition, 2017, <https://arxiv.org/abs/1610.02357>.

Несмотря на неоднократные обращения к NVIDIA, раздельные свертки по глубине так и не получили того же уровня программной и аппаратной оптимизации, что и обычные. В результате скорости их выполнения остаются примерно одинаковыми, несмотря на то что параметров и операций с плавающей точкой у них квадратично меньше. И все же использование раздельных по глубине сверток остается хорошей идеей даже в отсутствие ускорения: меньшее количество параметров означает меньшую подверженность риску переобучения, а предположение о независимости каналов приводит к более быстрой сходимости модели и получению более надежных представлений.

Небольшое неудобство в одном случае может превратиться в непроходимую стену в другом: вся аппаратная и программная экосистема глубокого обучения оптимизирована для очень конкретного набора алгоритмов (в частности, сверточных сетей, обучаемых через обратное распространение) и любое отклонение от проторенных дорог обходится чрезвычайно дорого. Если вам доведется экспериментировать с альтернативными алгоритмами, такими как безградиентная оптимизация или спайковые нейронные сети (Spiking Neural Networks), то первые ваши параллельные реализации на C++ или CUDA будут на порядки медленнее старой доброй сверточной сети, и неважно, насколько умны и эффективны ваши идеи. Вам будет сложно убедить других исследователей принять ваш метод, даже если он окажется лучше.

Современное глубокое обучение является продуктом коэволюции оборудования, программного обеспечения и алгоритмов. Доступность графических процессоров NVIDIA и CUDA привела к первому успеху сверточных сетей, обучаемых через обратное распространение, что заставило NVIDIA оптимизировать свои аппаратные и программные технологии для этих алгоритмов. Это, в свою очередь, привело к консолидации исследовательского сообщества вокруг данных методов. Выбор же другого пути в настоящее время потребует многолетней реорганизации всей экосистемы.

### **9.3.5. Собираем все вместе: мини-модель с архитектурой Xception**

Вспомним принципы архитектуры сверточных сетей, с которыми вы уже познакомились:

- модель должна быть организована в повторяющиеся *блоки* слоев, обычно состоящие из нескольких сверточных слоев и слоя выбора максимального значения из соседних;
- количество фильтров в слоях должно увеличиваться с уменьшением размеров карт пространственных признаков;

- глубокие и узкие модели лучше широких и неглубоких;
- добавление остаточных связей в обход блоков слоев помогает обучать более глубокие сети;
- иногда полезно добавлять слои пакетной нормализации после сверточных слоев;
- иногда полезно заменить слои Conv2D слоями SeparableConv2D, более эффективными по параметрам.

Попробуем воплотить все эти идеи в модели с архитектурой, напоминающей уменьшенную версию архитектуры Xception, и применить ее для решения задачи распознавания кошек и собак из прошлой главы. Для загрузки данных и обучения модели мы повторно используем код из пункта 8.2.5, но заменим определение модели следующей сверточной сетью:

```

inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)                                     ← Настройки для механизма обогащения
                                                               данных остаются такими же

x = layers.Rescaling(1./255)(x)                                     ← Не забудьте масштабировать
                                                               входные данные!

x = layers.Conv2D(filters=32, kernel_size=5, use_bias=False)(x)    ←

→ for size in [32, 64, 128, 256, 512]:
    residual = x

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

    residual = layers.Conv2D(
        size, 1, strides=2, padding="same", use_bias=False)(residual)
    x = layers.add([x, residual])                                     ← В первоначальной модели мы использовали
                                                               слой Flatten перед слоем Dense. Здесь
                                                               применяется слой GlobalAveragePooling2D

    x = layers.GlobalAveragePooling2D()(x)                            ←
    x = layers.Dropout(0.5)(x)                                       ←
    outputs = layers.Dense(1, activation="sigmoid")(x)              ←
model = keras.Model(inputs=inputs, outputs=outputs)                  ← Так же, как в исходной
                                                               модели, мы добавили
                                                               слой прореживания для
                                                               регуляризации

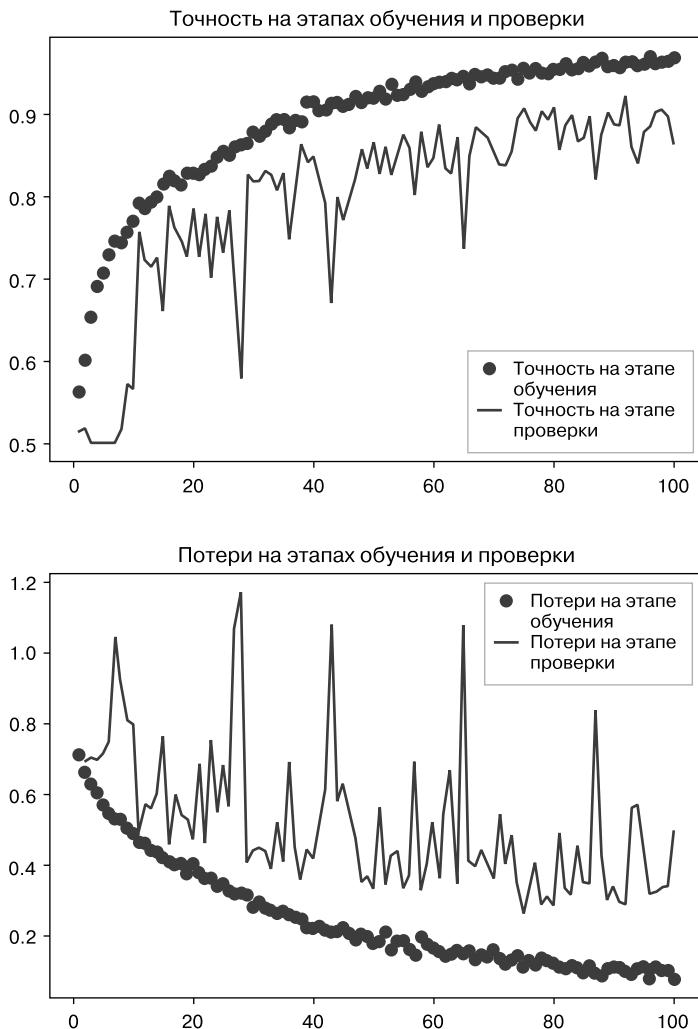
```

**Создать последовательность сверточных блоков с увеличением глубины признаков. Каждый блок включает два слоя раздельной свертки по глубине, два слоя пакетной нормализации, слой объединения с выбором максимального значения из соседних и остаточную связь в обход всего блока**

**Обратите внимание, что предположение о «практически полной независимости каналов признаков», лежащее в основе раздельной свертки, не выполняется для изображений RGB!**

Каналы красного, зеленого и синего цветов на самом деле сильно коррелируют в естественных изображениях. Поэтому первый слой в нашей модели — это обычный слой Conv2D. Слои SeparableConv2D будут использоваться потом

Эта сверточная сеть имеет 721 857 обучаемых параметров — меньше, чем в первоначальной модели, где обучаемых параметров было 991 041, — тем не менее качество ее прогнозов находится на аналогичном уровне. На рис. 9.11 показаны графики изменения точности и потерь на этапах обучения и проверки.



**Рис. 9.11.** Графики изменения метрик на этапах обучения и проверки для Xception-подобной архитектуры

Наша новая модель достигла точности 90,8 % на контрольных данных, что существенно лучше 83,5 % у первоначальной модели из предыдущей главы.

Как видите, следование рекомендациям по архитектуре оказывает значительное влияние на качество!

Чтобы увеличить точность прогнозирования еще больше, нужно провести систематическую настройку гиперпараметров архитектуры — эту тему мы подробно рассмотрим в главе 13. Пока мы пропустили этот шаг, так что предыдущая модель создана исключительно с использованием обсуждавшихся практик и (когда дело дошло до определения размера модели) небольшой доли интуиции.

Обратите внимание, что рекомендуемые приемы организации архитектуры моделей, представленные в этой главе, относятся к компьютерному зрению в целом, а не только к классификации изображений. Например, архитектура Xception используется как стандартная сверточная основа в DeepLabV3 — современном решении сегментации изображений<sup>1</sup>.

На этом мы завершаем вводное знакомство с передовыми приемами организации архитектур сверточных сетей. Опираясь на указанные принципы, можно разрабатывать весьма эффективные модели для широкого круга задач компьютерного зрения. Постепенно вы становитесь опытным специалистом в данной сфере. Чтобы еще больше углубить ваш опыт, затронем следующую важную тему: интерпретацию знаний, заключенных в модели.

## 9.4. ИНТЕРПРЕТАЦИЯ ЗНАНИЙ, ЗАКЛЮЧЕННЫХ В СВЕРТОЧНОЙ НЕЙРОННОЙ СЕТИ

Фундаментальной проблемой приложений компьютерного зрения является *интерпретируемость результатов*: почему классификатор решил, что конкретное изображение содержит холодильник, тогда как на нем присутствует только грузовик? Это особенно актуально для случаев, когда глубокое обучение используется как дополнение к человеческому опыту, например в медицинской визуализации. В заключение этой главы мы познакомимся с некоторыми приемами визуализации знаний, накопленных сверточными сетями, и интерпретации принимаемых ими решений.

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

<sup>1</sup> Chen L.-C. et al. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation // ECCV, 2018, <https://arxiv.org/abs/1802.02611>.

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 8.2. Для демонстрации двух других методов возьмем предварительно обученную модель Xception.

#### 9.4.1. 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 его называют *активацией*). Этот прием позволяет увидеть, как входные данные разлагаются на разные 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 8.2:

```
>>> from tensorflow import keras  
>>> model = keras.models.load_model(  
    "convnet_from_scratch_with_augmentation.keras")  
>>> model.summary()  
Model: "model_1"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[None, 180, 180, 3]	0
<hr/>		
sequential (Sequential)	(None, 180, 180, 3)	0
<hr/>		
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
<hr/>		

conv2d_5 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_6 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_5 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_7 (Conv2D)	(None, 41, 41, 128)	73856
max_pooling2d_6 (MaxPooling2D)	(None, 20, 20, 128)	0
conv2d_8 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_7 (MaxPooling2D)	(None, 9, 9, 256)	0
conv2d_9 (Conv2D)	(None, 7, 7, 256)	590080
flatten_1 (Flatten)	(None, 12544)	0
dropout (Dropout)	(None, 12544)	0
dense_1 (Dense)	(None, 1)	12545
<hr/>		
Total params:	991,041	
Trainable params:	991,041	
Non-trainable params:	0	

Далее выберем входное изображение кошки, не являющееся частью обучающего набора.

#### Листинг 9.6. Предварительная обработка единственного изображения

```
from tensorflow import keras
import numpy as np

img_path = keras.utils.get_file(
    fname="cat.jpg",
    origin="https://img-datasets.s3.amazonaws.com/cat.jpg")
```

Загрузить  
контрольное  
изображение

```
def get_img_array(img_path, target_size):
    img = keras.utils.load_img(
        img_path, target_size=target_size)
    array = keras.utils.img_to_array(img)
```

Открыть файл с изображением  
и изменить его размер

```
    array = np.expand_dims(array, axis=0)
    return array
```

←  
Преобразовать изображение  
в массив NumPy типа float32  
с формой (180, 180, 3)

```
img_tensor = get_img_array(img_path, target_size=(180, 180))

Добавить измерение для преобразования
массива в «пакет» с единственным образом.
Теперь он имеет форму (1, 180, 180, 3)
```

Отобразим исходное изображение (рис. 9.12).

**Листинг 9.7.** Отображение контрольного изображения

```
import matplotlib.pyplot as plt
plt.axis("off")
plt.imshow(img_tensor[0].astype("uint8"))
plt.show()
```

Для извлечения карт признаков, подлежащих визуализации, создадим модель Keras, которая принимает пакеты изображений и выводит активации всех сверточных и объединяющих слоев.



**Рис. 9.12.** Контрольное изображение кошки

**Листинг 9.8.** Создание экземпляра модели, возвращающей активации слоя

```
from tensorflow.keras import layers

layer_outputs = []
layer_names = []
for layer in model.layers:
    if isinstance(layer, (layers.Conv2D, layers.MaxPooling2D)):
        layer_outputs.append(layer.output)
        layer_names.append(layer.name)
activation_model = keras.Model(inputs=model.input, outputs=layer_outputs)
```

Извлечь выход всех слоев Conv2D и MaxPooling2D

Сохранить имена слоев для последующего использования

Создать модель, возвращающую выходы 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, который встречается вам на практике, с тех пор как вы познакомились с ними в главе 7: все представленные выше модели имели ровно один вход и один выход. В частности, данная модель имеет один вход и девять выходов: по одному на каждую активацию слоя.

**Листинг 9.9.** Использование модели для вычисления активаций слоев

```
activations = activation_model.predict(img_tensor) ← Вернет список с девятью массивами NumPy: по одному массиву на каждую активацию слоя
```

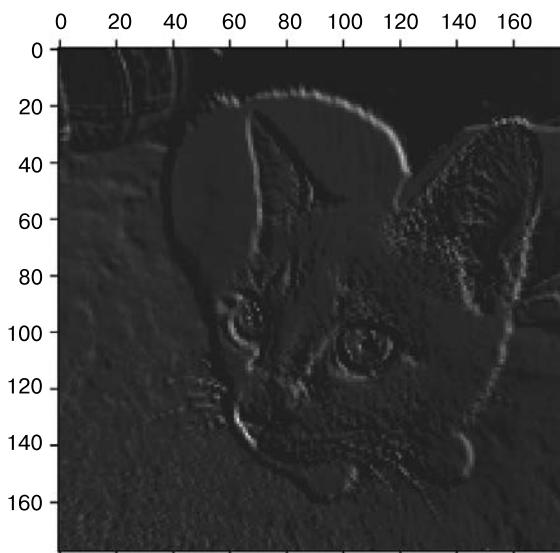
Возьмем для примера активацию первого сверточного слоя для входного изображения кошки:

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 178, 178, 32)
```

Это карта признаков  $178 \times 178$  с 32 каналами. Попробуем отобразить пятый канал активации первого слоя оригинальной модели (рис. 9.13).

**Листинг 9.10.** Визуализация пятого канала

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 5], cmap="viridis")
```



**Рис. 9.13.** Пятый канал активации первого слоя для контрольного изображения кошки

Похоже, что этот канал представляет собой диагональный детектор контуров — но имейте в виду, что у вас каналы могут отличаться, потому что обучение конкретных фильтров не является детерминированной операцией.

Теперь построим полную визуализацию всех активаций в сети (рис. 9.14). Для этого извлечем и отобразим каналы активации всех слоев, поместив результаты в одну большую сетку с изображениями.

### Листинг 9.11. Визуализация каждого канала для всех промежуточных активаций

```
images_per_row = 16
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1] | Активация слоя имеет форму
    size = layer_activation.shape[1] | (1, size, size, n_features)
    n_cols = n_features // images_per_row | Подготовить
    display_grid = np.zeros(((size + 1) * n_cols - 1, | пустую сетку
                            images_per_row * (size + 1) - 1)) | для отображения
    for col in range(n_cols): | всех каналов этой
        for row in range(images_per_row): | активации
            channel_index = col * images_per_row + row | Это единственный канал (признак)
            channel_image = layer_activation[0, :, :, channel_index].copy() |
            if channel_image.sum() != 0:
                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 + 1): (col + 1) * size + col,
                    row * (size + 1) : (row + 1) * size + row] = channel_image |
    scale = 1. / size | Отобразить
    plt.figure(figsize=(scale * display_grid.shape[1], | сетку для слоя
                scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.axis("off")
    plt.imshow(display_grid, aspect="auto", cmap="viridis") |

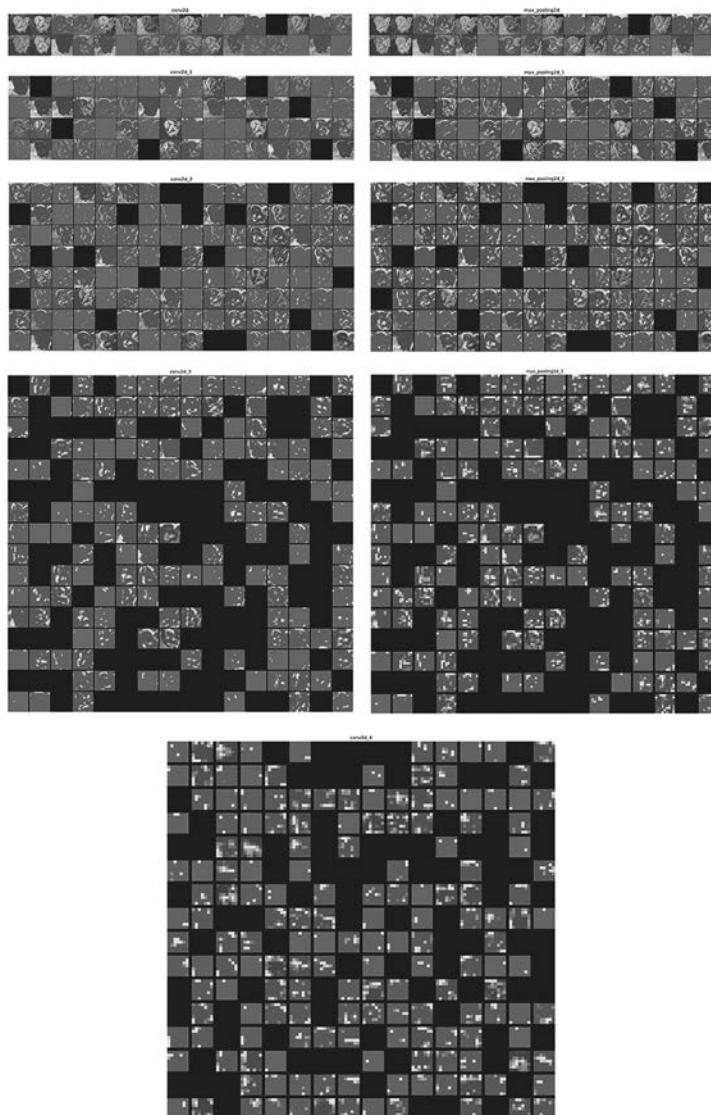
Поместить матрицу канала | Нормализовать значения канала, приведя их
в подготовленную пустую сетку | к диапазону [0, 255]. Все нулевые каналы остаются нулевыми
```

Вот 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.
- 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 «коша-

чий глаз». Высокоуровневые представления несут все меньше информации об исходном изображении и все больше — о классе изображения.

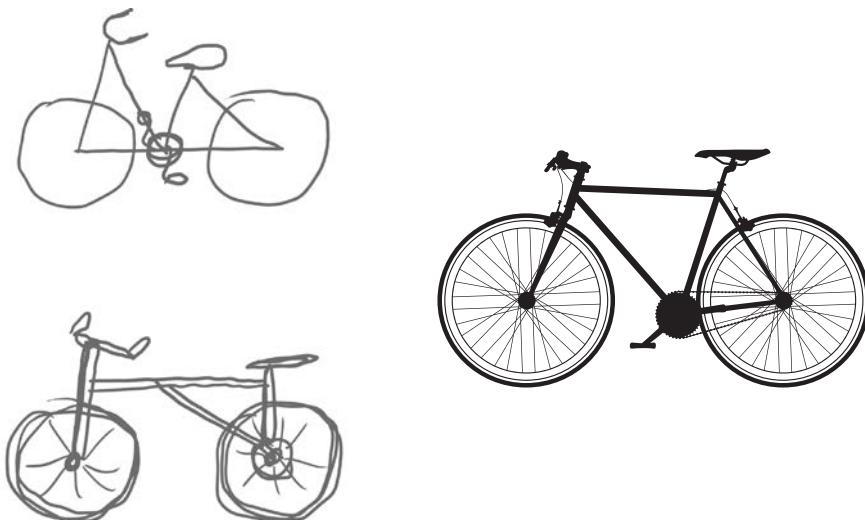
- Разреженность активаций увеличивается с глубиной слоя: в первом слое почти все фильтры активируются исходным изображением, но в последующих слоях все больше и больше остается пустых фильтров. Это означает, что шаблон, соответствующий фильтру, не обнаруживается в исходном изображении.



**Рис. 9.14.** Все каналы всех активаций слоев для контрольного изображения кошки

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 *конвойер 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, скорее всего, вам не удастся получить более или менее правильное изображение даже притом, что вы могли видеть велосипеды тысячи раз (см. примеры на рис. 9.15). 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. 9.15.** Слева: попытки нарисовать велосипед по памяти. Справа: так должен был бы выглядеть схематичный рисунок велосипеда

## 9.4.2. 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.

Попробуем проделать это с фильтрами модели Xception, обученной на наборе данных ImageNet. 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. Это будет наш второй пример реализации цикла низкоуровневого градиентного спуска с использованием объекта `GradientTape` (первый был показан в главе 2).

Для начала создадим экземпляр модели Xception, загрузив веса, полученные при обучении на наборе данных ImageNet.

### Листинг 9.12. Создание экземпляра сверточной основы модели Xception

```
model = keras.applications.xception.Xception(  
    weights="imagenet",  
    include_top=False)
```

Слой классификации в этом варианте  
использования модели не нужны,  
поэтому отключим их

Нас интересуют сверточные слои модели — `Conv2D` и `SeparableConv2D`. Но, чтобы получить их результаты, нужно знать имена слоев. Давайте выведем эти имена в порядке увеличения глубины.

### Листинг 9.13. Вывод имен всех сверточных слоев в модели Xception

```
for layer in model.layers:  
    if isinstance(layer, (keras.layers.Conv2D, keras.layers.SeparableConv2D)):  
        print(layer.name)
```

Обратите внимание, что все слои `SeparableConv2D` получили имена вида `block6_sepconv1`, `block7_sepconv2` и т. д. Модель Xception организована в блоки, каждый из которых содержит несколько сверточных слоев.

Теперь создадим вторую модель, которая вернет выходные данные определенного слоя, — модель *экстрактора признаков*. Поскольку наша модель создается с применением функционального API, ее можно проверить: запросить `output` одного из слоев и повторно использовать его в новой модели. Нет необходимости копировать весь код Xception.

**Листинг 9.14.** Создание модели экстрактора признаков

```

Эту строку можно заменить именем любого
слоя в сверточной основе Xception

layer_name = "block3_sepconv1" ← Объект слоя, который
layer = model.get_layer(name=layer_name) ← нас интересует
feature_extractor = keras.Model(inputs=model.input, outputs=layer.output) ←
                                                 Мы используем model.input и layer.output для создания
                                                 модели, которая возвращает выход целевого слоя

```

Чтобы использовать эту модель, просто передайте ей некоторые входные данные (обратите внимание, что модель Xception требует предварительной обработки входных данных с помощью функции `keras.applications.xception.preprocess_input`).

**Листинг 9.15.** Использование экстрактора признаков

```

activation = feature_extractor(
    keras.applications.xception.preprocess_input(img_tensor)
)

```

Воспользуемся нашей моделью экстрактора признаков, чтобы определить функцию, возвращающую скалярное значение, которое количественно определяет, насколько данное входное изображение активирует данный фильтр в слое. Это функция потерь, которую мы максимизируем в процессе градиентного восхождения:

```

import tensorflow as tf
def compute_loss(image, filter_index): ← Функция потерь принимает
    activation = feature_extractor(image) тензор с изображением и индекс
    filter_activation = activation[:, 2:-2, 2:-2, filter_index] ← фильтра (целое число)
    return tf.reduce_mean(filter_activation)

```

Вернуть среднее значений  
активации для фильтра

Обратите внимание: исключая из вычисления  
потерь пиксели, лежащие на границах,  
мы избегаем пограничных артефактов;  
в данном случае мы отбрасываем первые  
два пикселя по сторонам активации

**РАЗНИЦА МЕЖДУ MODEL.PREDICT(X) И MODEL(X)**

В предыдущей главе для извлечения признаков мы использовали `predict(x)`. Здесь мы берем `model(x)`. Почему?

Оба вызова, `y = model.predict(x)` и `y = model(x)`, где `x` — массив входных данных, подразумевают «запуск модели с исходными данными `x` и получение результата `y`». Но в обоих случаях данная формулировка обозначает не совсем одно и то же.

Метод `predict()` перебирает данные (при желании можно указать размер пакета, выполнив вызов `predict(x, batch_size=64)`) и извлекает массив NumPy с выходными данными. Схематично его реализацию можно представить так:

```
def predict(x):
    y_batches = []
    for x_batch in get_batches(x):
        y_batch = model(x).numpy()
        y_batches.append(y_batch)
    return np.concatenate(y_batches)
```

Таким образом, вызовы `predict()` могут обрабатывать очень большие массивы. Между тем `model(x)` выполняет обработку в памяти и не масштабируется. В то же время `predict()` не дифференцируется: нельзя получить его градиент, вызывая в контексте `GradientTape`.

Если нужно получить градиенты вызовов модели, используйте `model(x)`; если нужен только результат применения модели — берите `predict()`. Иными словами, `predict()` будет полезен во всех случаях, кроме реализации цикла низкоуровневого градиентного спуска (как сейчас).

Давайте реализуем функцию градиентного восхождения с помощью `GradientTape`. Обратите внимание, что для ускорения мы будем использовать декоратор `@tf.function`.

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.

#### Листинг 9.16. Максимизация потерь методом стохастического градиентного восхождения

<p>Явно передать для наблюдения тензор с изображением, потому что это не объект <code>Variable</code> (автоматически под наблюдение попадают только объекты <code>Variable</code>)</p> <pre>@tf.function def gradient_ascent_step(image, filter_index, learning_rate):     with tf.GradientTape() as tape:         tape.watch(image)</pre>	<p>Вычислить скаляр потерь, показывающий, насколько текущее изображение активирует фильтр</p>
<pre>        loss = compute_loss(image, filter_index)</pre>	<p>Вычислить градиенты потерь по отношению к изображению</p>
<pre>        grads = tape.gradient(loss, image)</pre>	<p>Применить «трюк нормализации градиента»</p>
<pre>        grads = tf.math.l2_normalize(grads)</pre>	
<pre>        image += learning_rate * grads</pre>	
<p>Вернуть обновленное изображение, чтобы дать возможность вызывать эту функцию в цикле</p>	<p>Немного сдвинуть изображение в направлении наибольшей активации целевого фильтра</p>

Теперь у нас есть все необходимые элементы. Объединим их в функцию на Python, которая будет принимать имя слоя и индекс фильтра и возвращать тензор, представляющий собой шаблон, который максимизирует активацию заданного фильтра.

#### Листинг 9.17. Функция, которая генерирует изображение, представляющее фильтр

```
img_width = 200
img_height = 200
def generate_filter_pattern(filter_index):
    iterations = 30
    learning_rate = 10.
    image = tf.random.uniform(
        minval=0.4,
        maxval=0.6,
        shape=(1, img_width, img_height, 3))
    for i in range(iterations):
        image = gradient_ascent_step(image, filter_index, learning_rate)
    return image[0].numpy()
```

Полученный тензор с изображением — это массив с формой (200, 200, 3) и вещественными значениями, которые могут быть нецелочисленными, в диапазоне [0, 255]. Поэтому нужно дополнительно его обработать, чтобы превратить в изображение, пригодное для показа. Сделаем это с помощью простой вспомогательной функции.

#### Листинг 9.18. Вспомогательная функция для преобразования тензора в изображение

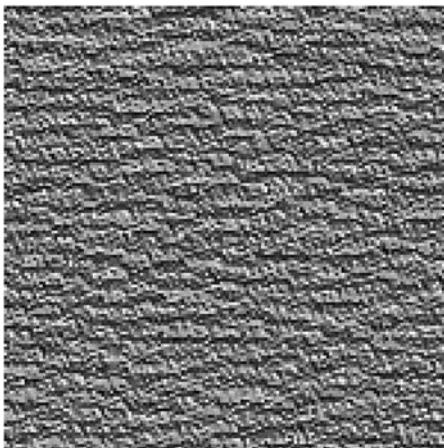
```
def deprocess_image(image):
    image -= image.mean()
    image /= image.std()
    image *= 64
    image += 128
    image = np.clip(image, 0, 255).astype("uint8")
    image = image[25:-25, 25:-25, :]
    return image
```

Взглянем на получившееся изображение (рис. 9.16):

```
>>> plt.axis("off")
>>> plt.imshow(deprocess_image(generate_filter_pattern(filter_index=2)))
```

Похоже, что фильтр с индексом 0 в слое `block3_sepconv1` отвечает за узор из горизонтальных линий, немного похожий на водную гладь или на мех.

А теперь самое интересное: мы можем визуализировать все фильтры в слое или даже все фильтры во всех слоях модели.



**Рис. 9.16.** Шаблон, на который второй канал в слое block3\_sepconv1 дает максимальный отклик

#### Листинг 9.19. Создание сетки со всеми шаблонами откликов в слое

```

all_images = []
for filter_index in range(64):           ← Сгенерировать и сохранить изображения
    print(f"Processing filter {filter_index}")
    image = deprocess_image(
        generate_filter_pattern(filter_index)
    )
    all_images.append(image)             ← Подготовить чистый холст для добавления
                                         изображений фильтров

margin = 5                                ←
n = 8                                     ←
cropped_width = img_width - 25 * 2          ←
cropped_height = img_height - 25 * 2         ←
width = n * cropped_width + (n - 1) * margin
height = n * cropped_height + (n - 1) * margin
stitched_filters = np.zeros((width, height, 3))

for i in range(n):                         ← Заполнить изображение
    for j in range(n):                     ← сохраненными фильтрами
        image = all_images[i * n + j]
        stitched_filters[                ←
            row_start = (cropped_width + margin) * i
            row_end = (cropped_width + margin) * i + cropped_width
            column_start = (cropped_height + margin) * j
            column_end = (cropped_height + margin) * j + cropped_height

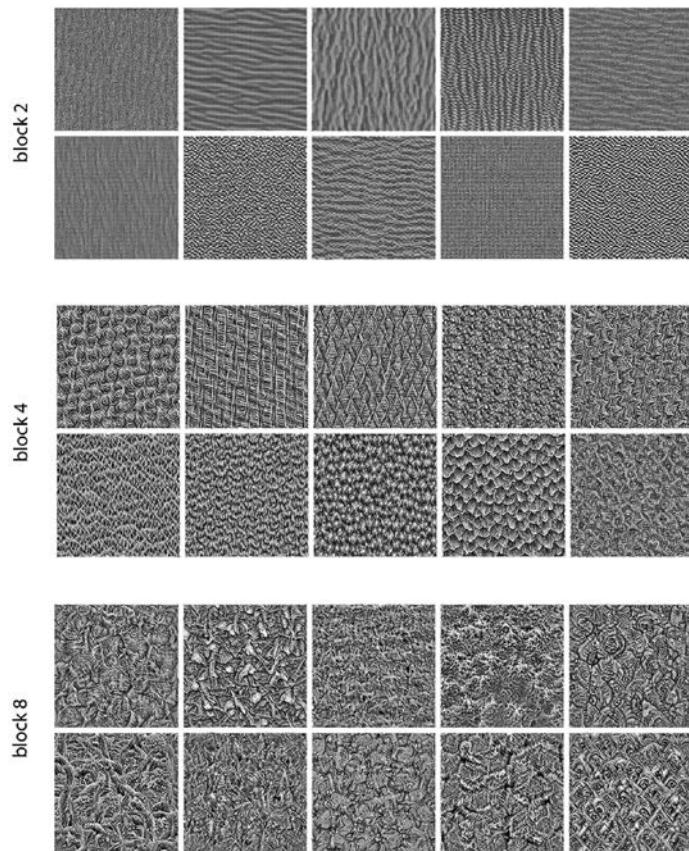
            stitched_filters[           ← Сохранить холст
                row_start: row_end,
                column_start: column_end, :] = image
                                         ← на диск

keras.utils.save_img(
    f"filters_for_layer_{layer_name}.png", stitched_filters)

```

Эти визуальные представления фильтров (рис. 9.17) могут многое рассказать о том, как слои сверточной нейронной сети видят мир: каждый слой в сети обуславливает 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 кодируют простые направленные контуры и цвета (или в некоторых случаях цветные контуры);
- фильтры из слоев чуть выше (таких как `block4_sepconv1`) 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.



**Рис. 9.17.** Некоторые шаблоны фильтров из слоев `block2_sepconv1`, `block4_sepconv1` и `block8_sepconv1`

### 9.4.3. Визуализация тепловых карт активации класса

В этом пункте 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 (данная предметная область называется *интерпретируемостью модели*). 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 сгенерировать 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».

Продемонстрируем этот прием с использованием предварительно обученной сети Xception.

#### Листинг 9.20. Загрузка предварительно обученной сети Xception

```
model = keras.applications.xception.Xception(weights="imagenet") ←  
        Обратите внимание на то, что мы добавили сверху полно связанный  
        классификатор; во всех предыдущих случаях мы отбрасывали его
```

Рассмотрим фотографию двух африканских слонов на рис. 9.18, на которой изображены самка и ее слоненок, прогуливающиеся по саванне. Преобразуем

<sup>1</sup> Selvaraju R. R. et al. arXiv, 2017, <https://arxiv.org/abs/1610.02391>.

этую фотографию в форму, которую сможет прочитать модель Xception. Модель обучена на изображениях размерами  $299 \times 299$ , предварительно обработанных в соответствии с правилами, реализованными в функции `keras.applications.xception.preprocess_input`, а потому мы также должны привести фотографию к размерам  $299 \times 299$ , преобразовать ее в тензор NumPy с числами типа `float32` и применить правила предварительной обработки.

**Листинг 9.21.** Предварительная обработка входного изображения для передачи в модель Xception

```
img_path = keras.utils.get_file(  
    fname="elephant.jpg",  
    origin="https://img-datasets.s3.amazonaws.com/elephant.jpg") ← Загрузка изображения  
def get_img_array(img_path, target_size): ← Изображение 299 × 299 в формате  
    img = keras.utils.load_img(img_path, target_size=target_size) ← Python Imaging Library (PIL)  
    array = keras.utils.img_to_array(img) ← Добавление размерности для преобразования  
    array = np.expand_dims(array, axis=0) ← массива в пакет с формой (1, 299, 299, 3)  
    array = keras.applications.xception.preprocess_input(array) ←  
    return array  
  
img_array = get_img_array(img_path, target_size=(299, 299))  
  
Массив NumPy с числами типа float32,  
имеющий форму (299, 299, 3) ← Предварительная обработка пакета  
                                (нормализация каналов цвета)
```

Теперь можно передать изображение в предварительно обученную сеть и декодировать полученный вектор в удобочитаемый формат:

```
>>> preds = model.predict(img_array)  
>>> print(keras.applications.xception.decode_predictions(preds, top=3)[0])  
[("n02504458", "African_elephant", 0.8699266),  
 ("n01871265", "tusker", 0.076968715),  
 ("n02504013", "Indian_elephant", 0.02353728)]
```

Вот первые три прогнозируемых класса для данного изображения:

- африканский слон (с вероятностью 87 %);
- кабан-секач (с вероятностью 7 %);
- индийский слон (с вероятностью 2 %).

Сеть распознала на изображении неопределенное количество африканских слонов. Элемент 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.



**Рис. 9.18.** Контрольная фотография африканских слонов

Прежде всего подготовим модель, отображающую входное изображение в активации последнего сверточного слоя.

**Листинг 9.22.** Подготовка модели, возвращающей вывод последнего сверточного слоя

```
last_conv_layer_name = "block14_sepconv2_act"
classifier_layer_names = [
    "avg_pool",
    "predictions",
]
last_conv_layer = model.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

Затем создадим модель, отображающую активации последнего сверточного слоя, чтобы получить прогнозы классов.

**Листинг 9.23.** Повторное применение классификатора к результату последнего сверточного слоя

```
classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
x = classifier_input
for layer_name in classifier_layer_names:
    x = model.get_layer(layer_name)(x)
classifier_model = keras.Model(classifier_input, x)
```

Вычислим градиент для наиболее вероятного класса входного изображения с учетом активаций последнего сверточного слоя.

**Листинг 9.24.** Получение градиентов для наиболее вероятного класса

```
import tensorflow as tf
with tf.GradientTape() as tape:
    last_conv_layer_output = last_conv_layer_model(img_array)
    tape.watch(last_conv_layer_output)
    preds = classifier_model(last_conv_layer_output)
    top_pred_index = tf.argmax(preds[0])
    top_class_channel = preds[:, top_pred_index]

grads = tape.gradient(top_class_channel, last_conv_layer_output) ←
```

Вычисление активаций последнего сверточного слоя и передача их для наблюдения объекту GradientTape

Канал активации, соответствующий наиболее вероятному классу

Градиент наиболее вероятного класса согласно выходной карте признаков последнего сверточного слоя

Теперь применим объединение и взвесим по важности тензор градиентов, чтобы получить тепловую карту активации класса.

**Листинг 9.25.** Объединение и взвешивание по важности тензора градиентов

Вектор, каждый элемент которого представляет среднюю интенсивность градиента для данного канала. Он количественно оценивает важность каждого канала для наиболее вероятного класса

```
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2)).numpy() ←
last_conv_layer_output = last_conv_layer_output.numpy()[0]
for i in range(pooled_grads.shape[-1]):
    last_conv_layer_output[:, :, i] *= pooled_grads[i] ←
heatmap = np.mean(last_conv_layer_output, axis=-1) ←
```

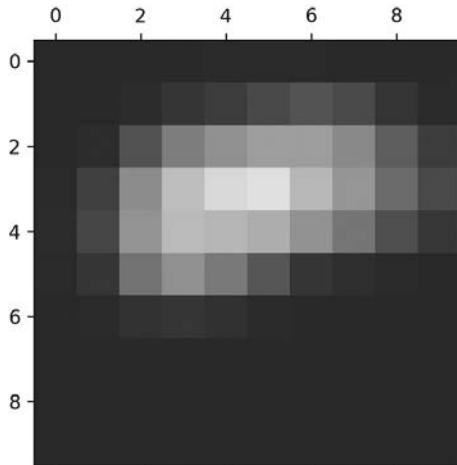
Умножить каждый канал в выводе последнего сверточного слоя на оценку «важность этого канала»

Среднее для каналов в полученной карте признаков — это тепловая карта активации класса

Для нужд визуализации нормализуем тепловую карту, приведя значения в ней к диапазону от 0 до 1. Результат показан на рис. 9.19.

**Листинг 9.26.** Заключительная обработка тепловой карты

```
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)
```

**Рис. 9.19.** Тепловая карта активации для отдельного класса

В заключение сгенерируем изображение, наложив тепловую карту на фотографию слонов (рис. 9.20).

**Листинг 9.27.** Наложение тепловой карты на исходное изображение

```
import matplotlib.cm as cm

img = keras.utils.load_img(img_path) | Загрузка исходного
img = keras.utils.img_to_array(img) | изображения

heatmap = np.uint8(255 * heatmap) ← Масштабирование тепловой
jet = cm.get_cmap("jet") | карты в диапазон 0–255
jet_colors = jet(np.arange(256))[:, :3] | Использование «струйной» (jet)
jet_heatmap = jet_colors[heatmap] | цветовой карты для раскрашивания
                                         | тепловой карты

jet_heatmap = keras.utils.array_to_img(jet_heatmap) | Создание изображения
jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0])) | с раскрашенной
jet_heatmap = keras.utils.img_to_array(jet_heatmap) | тепловой картой

superimposed_img = jet_heatmap * 0.4 + img | Наложение
superimposed_img = keras.utils.array_to_img(superimposed_img) | тепловой карты
                                                               | на оригинальную
                                                               | фотографию с уровнем
                                                               | прозрачности
                                                               | тепловой карты 40 %

save_path = "elephant_cam.jpg" | Сохранение полученного
superimposed_img.save(save_path) | изображения
```



**Рис. 9.20.** Оригинальная фотография с наложенной тепловой картой активации класса «африканский слон»

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

- Есть три основные задачи в сфере компьютерного зрения, которые можно решать с помощью глубокого обучения: классификация изображений, сегментация изображений и обнаружение объектов.
- Передовые методы организации архитектур сверточных сетей помогут получить максимальную отдачу от создаваемых моделей. К их числу относятся: использование остаточных связей, пакетная нормализация и раздельные свертки по глубине.
- Представления, получаемые сверточными нейронными сетями, легко поддаются исследованию, а значит, такие сети не являются черными ящиками!
- Вы научились визуализировать фильтры, полученные сверточной нейронной сетью, а также тепловые карты активации классов.

# 10

## *Глубокое обучение на временных последовательностях*

---

### **В этой главе**

- ✓ Примеры задач машинного обучения на временных последовательностях.
- ✓ Рекуррентные нейронные сети (Recurrent Neural Networks, RNN).
- ✓ Пример применения рекуррентных сетей для прогнозирования погоды.
- ✓ Улучшенные методы использования RNN.

### **10.1. РАЗНЫЕ ВИДЫ ВРЕМЕННЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ**

*Временные последовательности* (или временные ряды) — это любые данные, полученные путем измерений через регулярные промежутки времени, например стоимость акций в конце каждого дня, почасовое потребление электроэнергии в городе или еженедельные продажи в магазине. Временные последовательности присутствуют повсюду: и в природных явлениях (сейсмическая активность, изменение численности популяций рыб в реке или погода в определенном месте), и в закономерностях человеческой деятельности (посещаемость веб-сайта, изменение ВВП страны или операции с кредитной картой). В отличие от типов данных, с которыми вы сталкивались до сих пор, для работы с временными

последовательностями вы должны быть знакомы с понятием *динамики* системы — ее регулярности, наличия периодических циклов, временных закономерностей и внезапных всплесков.

Наиболее распространенной задачей, связанной с временными последовательностями, вне всяких сомнений, является *прогнозирование*: предсказание того, что произойдет дальше. Прогнозирование потребления электроэнергии на несколько часов вперед, чтобы предпринять необходимые меры; прогнозирование доходов на несколько месяцев вперед, чтобы спланировать бюджет; прогнозирование погоды на несколько дней вперед, чтобы скорректировать свои планы. Основное внимание в этой главе будет уделено прогнозированию, но вообще временные последовательности можно использовать для решения многих других задач.

- *Классификация* — присвоение временным последовательностям одной или нескольких категорий. Например, определение по временной активности посетителя веб-сайта, является он ботом или человеком.
- *Обнаружение событий* — определение момента наступления некоторого ожидаемого события в непрерывном потоке данных. В частности, широкое практическое применение получило «обнаружение горячих слов», когда модель распознает в аудиопотоке такие высказывания, как «о'кей, Google» или «привет, Алекса».
- *Обнаружение аномалий* — любых необычных событий в непрерывном потоке данных. Странная активность в корпоративной сети, например, может быть признаком действий злоумышленника. Необычные показания датчиков на производственной линии могут служить сигналом к вмешательству. Для определения аномалий обычно используется метод обучения без учителя, ведь часто нам неизвестно, какие события будут представлять интерес, поэтому обучить модель на конкретных примерах невозможно.

При работе с временными последовательностями вам встретится широкий спектр способов представления данных в разных предметных областях. Возможно, вы уже слышали о *преобразовании Фурье* — выражении последовательности значений в виде наложения волн разной частоты. Преобразование Фурье может очень пригодиться для предварительной обработки любых данных, характеризующихся своими циклами и колебаниями (например, для обработки звука, колебаний каркаса небоскреба или электромагнитных волн вашего мозга). В контексте глубокого обучения анализ Фурье (или связанный с ним мел-частотный анализ) и другие представления, зависящие от предметной области, могут быть полезны для проектирования признаков, то есть в качестве способа подготовки данных перед обучением модели. Однако мы не будем их рассматривать в этой книге и сосредоточимся на моделировании.

Далее в главе вы познакомитесь с рекуррентными нейронными сетями (Recurrent Neural Networks, RNN) и узнаете, как применять их для прогнозирования временных последовательностей.

## 10.2. ПРИМЕР ПРОГНОЗИРОВАНИЯ ТЕМПЕРАТУРЫ

Все дальнейшие примеры в этой главе будут нацелены на решение одной задачи: прогнозирование температуры на ближайшие 24 часа с помощью временной последовательности ежечасных измерений атмосферного давления и влажности, зарегистрированных в недавнем прошлом набором датчиков на крыше здания. Как вы убедитесь, это довольно сложная задача!

Мы покажем, какие принципиальные отличия имеют временные последовательности от наборов данных, с которыми вы сталкивались до сих пор. Вы увидите, что плотно связанные и сверточные сети плохо подходят для обработки таких наборов данных, но с ними блестящие спрашивается модели машинного обучения другого вида — рекуррентные нейронные сети.

Мы будем использовать временные последовательности данных о погоде, записанных на гидрометеорологической станции в Институте биогеохимии Макса Планка в Йене, Германия<sup>1</sup>. В этот набор данных включены замеры 14 разных характеристик (таких как температура, атмосферное давление, влажность, направление ветра и т. д.), выполнявшиеся каждые 10 минут в течение нескольких лет. Сбор данных был начат в 2003 году, но в этот пример включены только данные за 2009–2016 годы.

Давайте загрузим и распакуем архив с данными:

```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip  
!unzip jena_climate_2009_2016.csv.zip
```

А потом посмотрим, что у нас имеется.

**Листинг 10.1.** Обзор набора метеорологических данных Jena

```
import os  
fname = os.path.join("jena_climate_2009_2016.csv")  
  
with open(fname) as f:  
    data = f.read()  
  
lines = data.split("\n")  
header = lines[0].split(",")  
lines = lines[1:]  
print(header)  
print(len(lines))
```

---

<sup>1</sup> Адам Эрикссон и Олаф Колле, [www.bgc-jena.mpg.de/wetter](http://www.bgc-jena.mpg.de/wetter).

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)",
  "Tp0t (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 массивы NumPy: один для температуры (в градусах Цельсия), а другой для остальных данных — признаков, которые будут использоваться для прогнозирования температуры в будущем. Обратите внимание, что мы отбросили столбец **Date Time** (Дата и время).

### Листинг 10.2. Преобразование данных

```
import numpy as np
temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")][1:]]           Столбец 1 сохраняется
    temperature[i] = values[1]                                     в массиве temperature
    raw_data[i, :] = values[:]                                     ← Все остальные столбцы (включая температуру)
                                                               сохраняются в массиве raw_data
```

На рис. 10.1 показан график изменения температуры (в градусах Цельсия) с течением времени. На этом графике, охватывающем восьмилетний период, ясно виден годовой цикл изменения температуры.

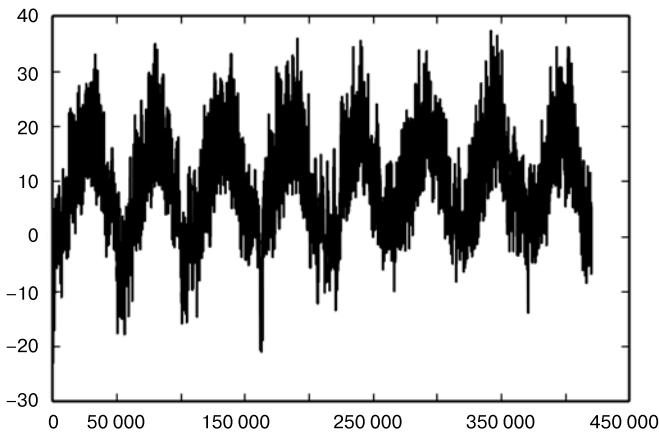
### Листинг 10.3. Создание графика изменения температуры

```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

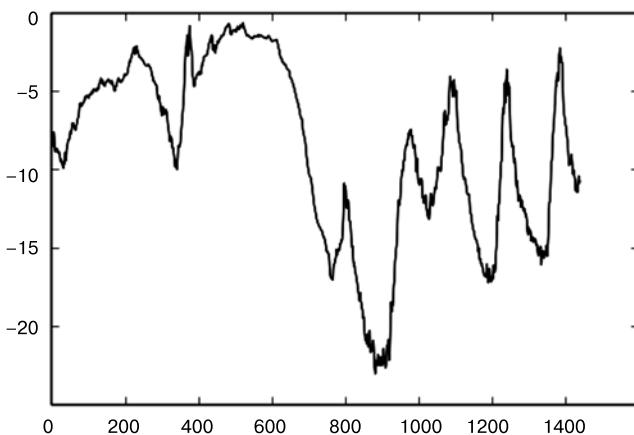
На рис. 10.2 показан более короткий график изменения температуры — за первые десять дней. Поскольку данные записываются каждые десять минут, за сутки накапливается  $24 \times 6 = 144$  замера.

**Листинг 10.4.** Создание графика изменения температуры по данным за первые десять дней

```
plt.plot(range(1440), temperature[:1440])
```



**Рис. 10.1.** График изменения температуры ( $^{\circ}\text{C}$ ), построенный по полному набору данных



**Рис. 10.2.** График изменения температуры ( $^{\circ}\text{C}$ ) по данным за первые десять дней

На этом графике можно видеть суточный цикл изменения температуры, который особенно четко наблюдается на отрезке, соответствующем последним четырем дням. Также отметьте, что этот десятидневный отрезок соответствует довольно холодному зимнему месяцу.

### ВСЕГДА ИЩИТЕ ЦИКЛИЧНОСТЬ В ВАШИХ ДАННЫХ

Цикличность — важное и очень распространенное свойство временных последовательностей. Независимо от того, какие данные вы рассматриваете — погоду, количество свободных мест на парковке в торговом центре, посещаемость веб-сайта, продажи в продуктовом магазине или количество шагов, подсчитанных шагомером, — вы неизменно будете наблюдать суточные и годовые циклы (данные, генерируемые человеком, также имеют недельные циклы). Поэтому при исследовании данных обязательно ищите подобные закономерности.

Если бы мы предсказывали среднюю температуру на следующий месяц по данным за несколько предыдущих месяцев, это 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.

В дальнейших экспериментах мы будем использовать первые 50 % данных для обучения, следующие 25 % — для проверки и последние 25 % — для контроля. При работе с временными последовательностями важно, чтобы проверочные и контрольные данные были более свежими, чем обучающие. Мы прогнозируем будущее на основе прошлого, а не наоборот, поэтому проверочная и контрольная выборки должны отражать это. Некоторые задачи оказываются значительно проще, если перевернуть ось времени!

#### Листинг 10.5. Вычисление количества образцов в каждой выборке

```
>>> num_train_samples = int(0.5 * len(raw_data))
>>> num_val_samples = int(0.25 * len(raw_data))
>>> num_test_samples = len(raw_data) - num_train_samples - num_val_samples
>>> print("num_train_samples:", num_train_samples)
>>> print("num_val_samples:", num_val_samples)
>>> print("num_test_samples:", num_test_samples)
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

### 10.2.1. Подготовка данных

Вот точная формулировка вопроса, на который нам нужно ответить в рамках задачи: можно ли предсказать температуру на следующие 24 часа по данным замеров, выполнявшихся один раз в час и охватывающих предыдущие пять дней?

Для начала преобразуем данные в формат, понятный нейронной сети. Это легко: они уже представлены в числовом виде, поэтому нам не придется их как-то векторизовать. Однако временные последовательности разных параметров в данных имеют разный масштаб (например, атмосферное давление, измеряемое в миллибарах, изменяется около значения 1000, а концентрация  $\text{CH}_2\text{O}$ , измеряемая в миллимолях на моль, колеблется около тройки). Мы должны нормализовать временные последовательности независимо друг от друга, чтобы все они состояли из небольших по величине значений примерно одинакового масштаба. В качестве обучающих данных мы будем использовать первые 210 225 замеров, поэтому вычислим среднее значение и стандартное отклонение только для этой части данных.

#### Листинг 10.6. Нормализация данных

```
mean = raw_data[:num_train_samples].mean(axis=0)
raw_data -= mean
std = raw_data[:num_train_samples].std(axis=0)
raw_data /= std
```

Затем создадим объект `Dataset`, возвращающий пакеты данных за последние пять дней вместе с целевой температурой через 24 часа в будущем. Поскольку образцы в наборе данных избыточны (образцы  $N$  и  $N + 1$  будут иметь много общего), весьма расточительно явно выделять каждый. Вместо этого мы будем генерировать образцы на лету, используя только исходные массивы `raw_data` и `temperature`.

Для этого мы могли бы написать генератор Python, но в Keras уже есть готовая утилита, которая как раз генерирует нужные нам образцы (`timeseries_dataset_from_array()`), поэтому можно сэкономить силы и время. Кстати, она подходит практически для любых задач прогнозирования временных последовательностей.

#### УТИЛИТА TIMESERIES\_DATASET\_FROM\_ARRAY()

Чтобы понять, что делает `timeseries_dataset_from_array()`, рассмотрим простой пример. Идея этой утилиты заключается в получении массива данных, составляющих временную последовательность (аргумент `data`), и возвращении окна, извлеченного из исходных временных последовательностей (будем называть их просто «последовательности»).

Например, для `data = [0 1 2 3 4 5 6]` и `sequence_length=3` утилита `timeseries_dataset_from_array()` сгенерирует следующие выборки: `[0 1 2]`, `[1 2 3]`, `[2 3 4]`, `[3 4 5]`, `[4 5 6]`.

Также в `timeseries_dataset_from_array()` можно передать аргумент `targets` (массив). Первый элемент `targets` должен соответствовать жела-

мой цели для первой последовательности, которая будет сгенерирована из массива данных. Поэтому при прогнозировании временных последовательностей цели должны быть таким же массивом, что и `data`, но со смещением на некоторую величину.

Например, для `data = [0 1 2 3 4 5 6 ...]` и `sequence_length=3` можно создать набор данных для прогнозирования следующего шага в последовательности, передав `targets = [3 4 5 6 ...]`. Давайте попробуем:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
for inputs, targets in dummy_dataset:
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

Генерируется отсортированный массив целых чисел от 0 до 9

Сгенерированные последовательности будут выбираться из [0 1 2 3 4 5]

Целью для последовательности, которая начинается с `data[N]`, должна быть `data[N + 3]`

Длина последовательности должна быть равна 3

Последовательности будут собираться в пакеты по две

Данный блок кода выведет следующий результат:

```
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
```

С помощью `timeseries_dataset_from_array()` мы создадим три набора данных: для обучения, для проверки и для контроля.

Для этого используем следующие параметры:

- `sampling_rate = 6` — наблюдения будут извлекаться по одному за каждый час, то есть по одному замеру из шести;
- `sequence_length = 120` — наблюдения будут возвращаться в прошлое на пять суток (120 часов);
- `delay = sampling_rate * (sequence_length + 24 - 1)` — целью для последовательности должна быть температура через 24 часа после конца последовательности.

При создании набора обучающих данных передадим `start_index = 0` и `end_index = num_train_samples`, чтобы использовать только первые 50 % данных.

Для получения проверочного набора данных передадим `start_index = num_train_samples` и `end_index = num_train_samples + num_val_samples`, чтобы включить в него следующие 25 % данных. Наконец, для получения контрольного набора данных передадим `start_index = num_train_samples + num_val_samples`, чтобы включить оставшиеся образцы.

**Листинг 10.7.** Создание наборов данных: обучающего, проверочного и контрольного

```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256

train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=0,
    end_index=num_train_samples)

val_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples,
    end_index=num_train_samples + num_val_samples)

test_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples + num_val_samples)
```

Каждый набор данных возвращает кортеж (**образцы**, **целевые значения**), где **образцы** — это пакет из 256 образцов, каждый из которых включает 120 последовательных замеров за каждый час, а **целевые значения** — соответствующий массив из 256 целевых температур. Обратите внимание, что образцы перемешиваются случайным образом, поэтому два соседних образца в пакете (например, `samples[0]` и `samples[1]`) не обязательно близки во времени.

**Листинг 10.8.** Исследование вывода, возвращаемого одним из экземпляров Dataset

```
>>> for samples, targets in train_dataset:  
>>>     print("samples shape:", samples.shape)  
>>>     print("targets shape:", targets.shape)  
>>>     break  
samples shape: (256, 120, 14)  
targets shape: (256,)
```

## 10.2.2. 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. Но иногда подобные базовые решения превзойти на удивление трудно.

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 10.9.** Оценка базового решения MAE

```
def evaluate_naive_method(dataset):  
    total_abs_err = 0.  
    samples_seen = 0  
    for samples, targets in dataset:
```

```

preds = samples[:, -1, 1] * std[1] + mean[1]
total_abs_err += np.sum(np.abs(preds - targets))
samples_seen += samples.shape[0]
return total_abs_err / samples_seen

```

print(f"Validation MAE: {evaluate\_naive\_method(val\_dataset):.2f}")  
print(f"Test MAE: {evaluate\_naive\_method(test\_dataset):.2f}")

Значения температуры находятся в столбце 1, поэтому `samples[:, -1, 1]` — это последний замер температуры во входной последовательности. Напомню: выше мы нормализовали наши данные, поэтому, чтобы получить температуру в градусах Цельсия, нужно ее денормализовать, умножив на стандартное отклонение и прибавив среднее значение

Это базовое решение обеспечивает среднюю абсолютную ошибку (MAE) 2,44 градуса Цельсия на проверочных данных и 2,62 градуса Цельсия — на контрольных. То есть, спрогнозировав температуру, которая будет через 24 часа, вы ошибетесь в среднем на два с половиной градуса. Не так плохо, но едва ли кто-то захочет запустить свою службу прогноза погоды, основанную на подобной эвристике. Давайте попробуем использовать наши знания в области глубокого обучения, чтобы улучшить результат.

### 10.2.3. 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 не-затратные 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 задач регрессии. В роли оценки потерь вместо средней абсолютной ошибки (MAE) мы будем использовать среднеквадратичную ошибку (mean squared error, MSE), поскольку, в отличие от MAE, функция MSE гладкая в районе нуля, что является полезным свойством для градиентного спуска. Но мы будем следить также и за величиной MAE, указав ее в качестве метрики в вызове `compile()`.

#### Листинг 10.10. Обучение и оценка полносвязной модели

```

from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)

```

```

outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
        save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
    
```

Использовать обратный вызов, чтобы сохранить лучшую модель

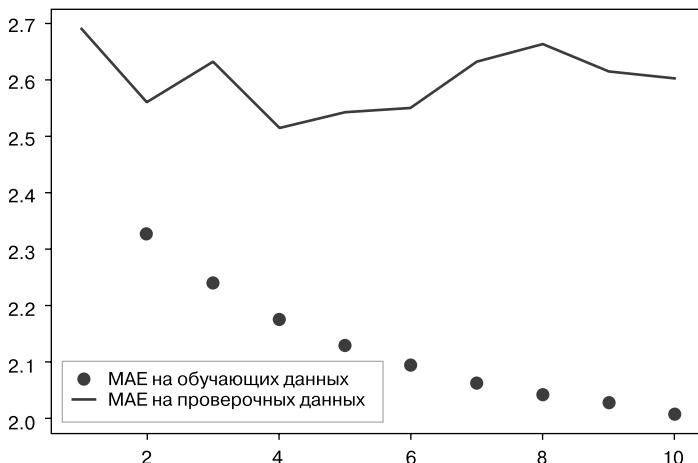
Загрузить лучшую модель и оценить ее на контрольных данных

Выведем кривые потерь на обучающих и проверочных данных (рис. 10.3).

#### Листинг 10.11. Вывод результатов

```

import matplotlib.pyplot as plt
loss = history.history["mae"]
val_loss = history.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "bo", label="MAE на обучающих данных")
plt.plot(epochs, val_loss, "b", label="MAE на проверочных данных")
plt.title("MAE на обучающих и проверочных данных")
plt.legend()
plt.show()
    
```



**Рис. 10.3.** Изменение средней абсолютной ошибки (MAE) простой полносвязной сети на обучающих и проверочных данных в задаче прогнозирования температуры по данным Jena

Некоторые значения потерять на этапе проверки близки к оценке 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. Наша полносвязная базовая модель — лишь одна из миллионов моделей, существующих в пространстве. Это как искать иголку в стоге сена. И факт, что в пространстве вашей гипотезы технически существует хорошее решение, не означает, что его удастся найти с помощью градиентного спуска.

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. Вот почему важно со всей тщательностью подходить к проектированию признаков и выбору соответствующей архитектуры: вы должны точно указать своей модели, что она должна искать.

#### **10.2.4. Попытка использовать одномерную сверточную модель**

К вопросу о выборе подходящей архитектуры: наши входные последовательности имеют ежедневные циклы, поэтому вполне возможно, что для их прогнозирования можно взять сверточную модель. Временная сверточная сеть может повторно использовать одни и те же представления в разные дни, так же как пространственная может повторно применять одни и те же представления в разных местах изображения.

Вы знаете, что существуют слои `Conv2D` и `SeparableConv2D`, которые видят входные данные через маленькие окна, перемещающиеся по двумерным сеткам. Существуют также одно- и даже трехмерные версии этих слоев: `Conv1D`, `SeparableConv1D` и `Conv3D`.<sup>1</sup> Слой `Conv1D` основан на одномерном окне, скользящем вдоль входных последовательностей, а слой `Conv3D` — на кубических окнах, скользящих вдоль входных объемов.

---

<sup>1</sup> Обратите внимание, что слой `SeparableConv3D` отсутствует в библиотеке, но не по каким-то теоретическим соображениям, а потому, что он просто пока не реализован.

То есть можно строить одномерные сверточные сети, аналогичные двумерным сверточным сетям. Они отлично подходят для прогнозирования на любых последовательных данных, для которых соблюдается предположение об инвариантности в отношении переноса (когда при перемещении окна вдоль последовательности его содержимое, независимо от местоположения окна, должно соответствовать тем же свойствам).

Давайте попробуем использовать такую сверточную модель в нашей задаче прогнозирования температуры. Выберем начальную длину окна 24, чтобы модель могла видеть данные сразу (в каждом цикле) за 24 часа. По мере уменьшения разрешения последовательностей (с использованием слоев `MaxPooling1D`) будем соответственно уменьшать размер окна:

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
        save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=callbacks)

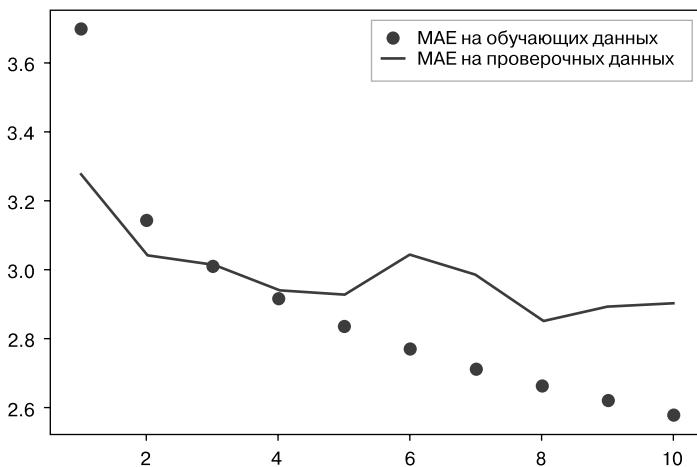
model = keras.models.load_model("jena_conv.keras")
print(f"Тестовая MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

На рис. 10.4 показаны кривые изменения метрик на обучающих и проверочных данных.

Как оказалось, данная модель прогнозирует температуру даже хуже, чем полно связная, достигнув средней абсолютной ошибки на проверочных данных всего 2,9 градуса, что далеко от уровня базовой модели. Почему так получилось? Есть две причины.

- Во-первых, данные о погоде не соответствуют полностью предположению о инвариантности в отношении переноса. Несмотря на наличие суточных циклов в данных, утренние данные обладают иными свойствами, чем вечерние илиочные. Данные о погоде являются инвариантными в отношении переноса только для очень конкретной временной шкалы.

- Во-вторых, порядок в данных о погоде имеет большое значение. Недавнее прошлое гораздо информативнее для предсказания температуры следующего дня, чем данные пятидневной давности. Одномерная сеть неспособна учесть этот факт. В частности, слои `MaxPooling1D` и `GlobalAveragePooling1D` уничтожают информацию о порядке.



**Рис. 10.4.** Изменение средней абсолютной ошибки (MAE) одномерной сверточной сети на обучающих и проверочных данных в задаче прогнозирования температуры по данным Jena

### 10.2.5. Первое базовое рекуррентное решение

Ни полносвязное, ни сверточное решение не дали хорошего результата — но это не означает, что машинное обучение неприменимо к данной задаче. В подходе на основе полносвязной модели первым действием мы уменьшили размерность временных последовательностей, устранив понятие времени из входных данных. Подход на основе сверточной сети одинаково обрабатывал каждый сегмент данных и применял операцию объединения, которая тоже удаляла информацию о порядке следования. Давайте посмотрим на эти данные как на то, чем они являются в действительности: последовательностью, в которой важны причина и следствие.

Для обучения на таких данных была создана специальная архитектура нейронных сетей: рекуррентные нейронные сети. Особенно большой популярностью пользуется слой долгой краткосрочной памяти (*long short term memory*, LSTM). Чуть ниже вы увидите, как работают данные нейронные сети, но прежде давайте опробуем слой LSTM.

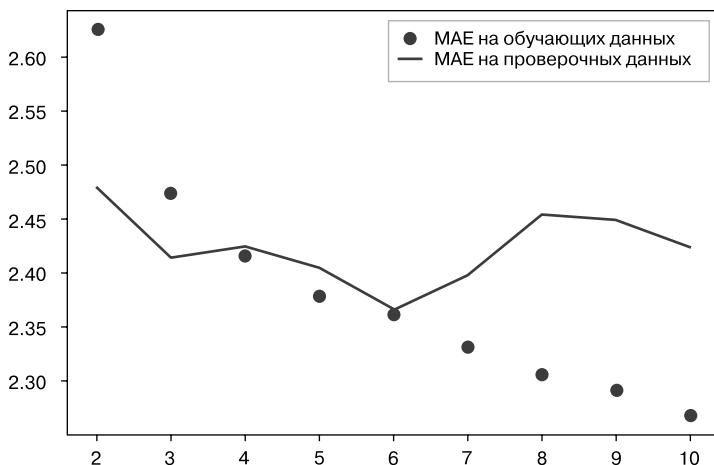
**Листинг 10.12.** Простая модель на основе слоя LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
        save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Тестовая MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

На рис. 10.5 показаны полученные результаты. Они оказались намного лучше! Мы достигли средней абсолютной ошибки на проверочных данных всего 2,36 градуса, а на контрольных данных — 2,55 градуса. Модель на основе LSTM смогла превзойти базовый уровень (хоть и ненамного), доказав, что машинное обучение способно справиться с этой задачей.



**Рис. 10.5.** Изменение средней абсолютной ошибки (MAE) модели со слоем LSTM на обучающих и проверочных данных в задаче прогнозирования температуры по данным Jena (обратите внимание, что на графике отсутствуют метрики для первой эпохи: причина в том, что величина MAE (7,75) после этой эпохи просто сильно искажает масштаб графика)

Но почему модель LSTM показала заметно лучший результат, чем полносвязная и сверточная модели? И можно ли еще больше повысить точность модели? Чтобы ответить на эти вопросы, познакомимся с рекуррентными нейронными сетями поближе.

### 10.3. РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ

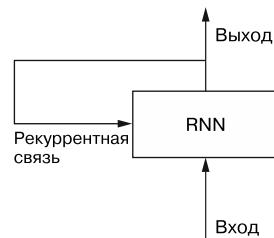
Главной характеристикой всех нейронных сетей, с которыми мы познакомились к данному моменту, таких как полносвязные и сверточные нейронные сети, является отсутствие памяти. Каждый вход обрабатывается ими независимо, без сохранения состояния между ними. Чтобы с помощью таких сетей обработать последовательность, или временной ряд, данных, необходимо передать в сеть всю последовательность целиком, преобразовав ее в единый пакет. Именно так мы поступили в предыдущем примере: мы объединили все данные за пять дней в один большой вектор и обработали его целиком. Такие сети называют *сетями прямого распространения* (feedforward networks).

С другой стороны, читая предложение в тексте, 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* (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. 10.6).

Сеть RNN сбрасывает состояние между обработкой двух разных, независимых последовательностей (таких как два образца из пакета), 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 простую сеть RNN с прямой передачей. Она будет принимать на входе последовательность векторов в виде двумерного тензора с формой (*временные\_интервалы*, *входные\_признаки*),



**Рис. 10.6.** Рекуррентная сеть — сеть с циклом

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 10.13. Реализация RNN в псевдокоде

```
state_t = 0 ← Состояние в момент t
for input_t in input_sequence:
    output_t = f(input_t, state_t) ← Цикл по последовательности
    state_t = output_t ← элементов
    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 полно связанный слой в сети прямого распространения.

#### Листинг 10.14. 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 10.15. Реализация сети RNN на основе NumPy

```
Число временных
интервалов во входной
последовательности
import numpy as np
timesteps = 100
input_features = 32
output_features = 64
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,))

Размерность пространства
входных признаков
Размерность пространства
выходных признаков
Входные данные: случайный
шум для простоты примера
Начальное состояние:
вектор с нулевыми
значениями элементов
Создание матриц
со случайными
весами
```

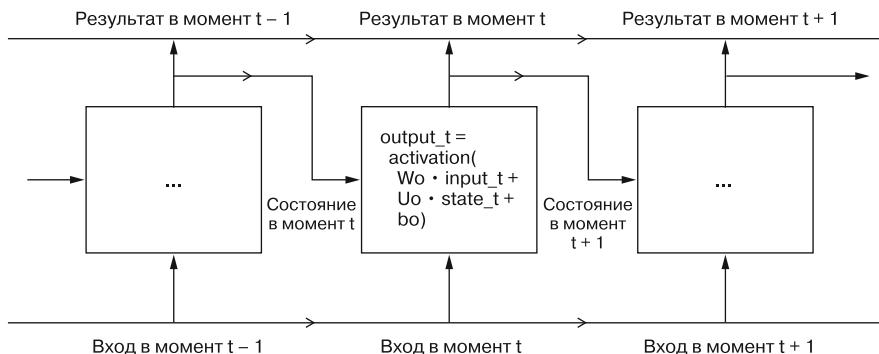
```

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.stack(successive_outputs, axis=0) ←
    Окончательный результат — двумерный тензор с формой
    (временные_интервалы, выходные_признаки)
    Обновление текущего состояния сети как подготовка
    к обработке следующего временного интервала
    Объединение входных данных с текущим состоянием (выходными
    данными на предыдущем шаге). Функция tanh используется для придачи
    нелинейности (здесь можно взять любую другую функцию активации)

```

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 данному определению, и этот пример — одна из простейших реализаций RNN. Рекуррентные сети характеризуются функцией, реализующей один шаг, такой как следующая, использованная в данном примере (рис. 10.7):

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```



**Рис. 10.7.** Простая рекуррентная сеть, развернутая во времени

### ПРИМЕЧАНИЕ

В этом примере конечный результат имеет вид двумерного тензора с формой (временные\_интервалы, выходные\_признаки), где каждый временному интервалу — 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.

### 10.3.1. Рекуррентный слой в Keras

Процессу, который мы только что реализовали с применением NumPy, соответствует фактический слой в Keras — слой `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`). Обратите внимание, что при вызове `Input()` с аргументом `shape` в элементе `временные_интервалы` можно передать значение `None` — это позволит сети обрабатывать последовательности произвольной длины.

**Листинг 10.16.** Слой сети RNN, способный обрабатывать последовательности любой длины

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

Это может пригодиться в моделях, предназначенных для обработки последовательностей переменной длины. Однако если все последовательности имеют одинаковую длину, лучше явно указать полную форму входных данных — это позволит методу `model.summary()` отображать информацию о длине выхода, что всегда полезно и в отдельных случаях дает возможность применению некоторых оптимизаций (см. примечание «О производительности RNN» далее в этой главе).

Все рекуррентные слои в Keras (`SimpleRNN`, `LSTM` и `GRU`) могут действовать в двух разных режимах: возвращать полные последовательности результатов для всех временных интервалов (трехмерный тензор с формой (`размер_пакета, временные_интервалы, выходные_признаки`)) или только последний результат для каждой входной последовательности (двумерный тензор с формой (`размер_пакета, выходные_признаки`)). Режимы управляются аргументом `return_sequences` конструктора. Рассмотрим пример, в котором используется слой `SimpleRNN` и возвращается результат только для последнего временного интервала.

**Листинг 10.17.** Слой сети RNN, возвращающий результат только для последнего интервала

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
>>> print(outputs.shape)
(None, 16)
```

Обратите внимание, что `return_sequences=False` —  
это значение по умолчанию

Следующий пример возвращает полную последовательность состояний.

**Листинг 10.18.** Слой рекуррентной сети, возвращающий полную последовательность результатов

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
>>> print(outputs.shape)
(120, 16)
```

Иногда, чтобы увеличить репрезентативность сети, полезно включить в модель несколько рекуррентных слоев, следующих друг за другом. В таких ситуациях все промежуточные слои должны возвращать полные последовательности результатов.

**Листинг 10.19.** Стек из нескольких слоев RNN

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

На практике, однако, слои `SimpleRNN` используются довольно редко — в большинстве случаев они слишком простые для реального применения. В частности, `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, Шмидхубером и Бенгио в начале 1990-х годов<sup>1</sup>.

К счастью, `SimpleRNN` — не единственный рекуррентный слой, доступный в Keras. Кроме него, имеются также слои `LSTM` и `GRU`, разработанные специально для решения подобных проблем.

Рассмотрим слой `LSTM`. Лежащий в его основе алгоритм долгой краткосрочной памяти (*long short-term memory*, LSTM) был разработан Хохрейтером и Шмидхубером в 1997 году<sup>2</sup>; он стал кульминацией их исследований проблемы затухания градиента.

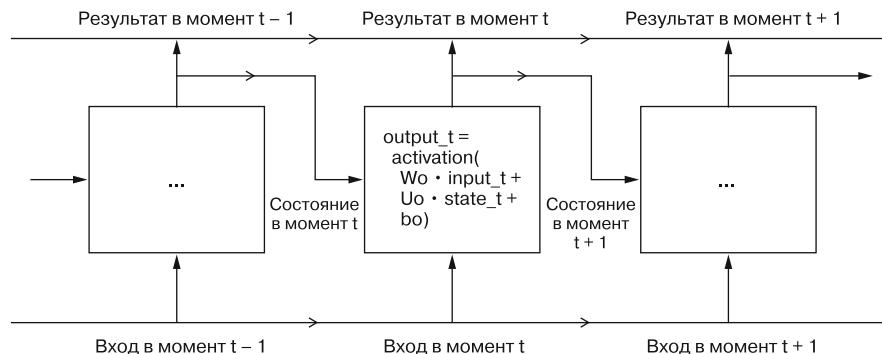
---

<sup>1</sup> См., например: *Bengio Y., Simard P., Frasconi P. Learning Long-Term Dependencies with Gradient Descent Is Difficult* // IEEE Transactions on Neural Networks 5, no. 2. 1994.

<sup>2</sup> *Hochreiter S., Schmidhuber J. Long Short-Term Memory* // Neural Computation 9, no. 8. 1997.

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. Данное решение напоминает *остаточные связи*, с которыми вы познакомились в главе 9; в действительности в его основе лежит практически та же самая идея.

Чтобы разобраться более детально, начнем с ячейки SimpleRNN (рис. 10.8). 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»).

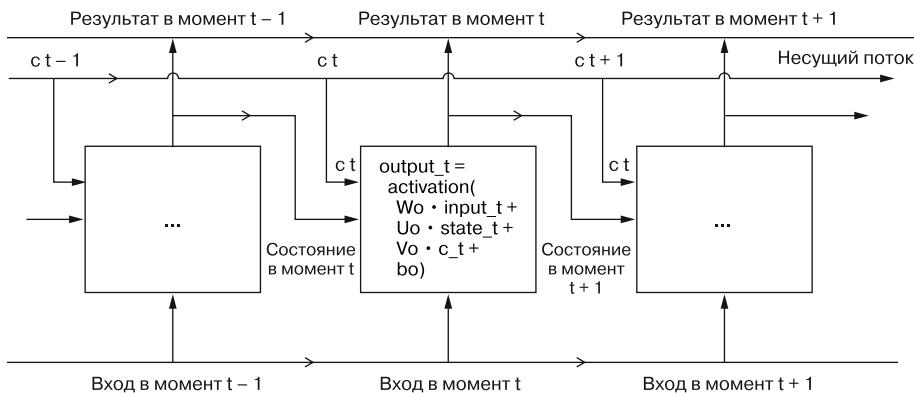


**Рис. 10.8.** Начальная точка слоя LSTM: слой SimpleRNN

Добавим в эту схему дополнительный поток данных, несущий информацию сквозь интервалы времени. Для его значений в разные интервалы времени будем использовать обозначение  $c_{\cdot t}$ , 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. 10.9). Пока все довольно просто.

А теперь о деталях способа вычисления следующего значения в несущем потоке данных: он основывается на трех разных преобразованиях, все три имеют форму ячейки SimpleRNN:

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```



**Рис. 10.9.** Переход от SimpleRNN к LSTM: добавление несущего потока

Однако эти преобразования имеют свои весовые матрицы, которые мы обозначим индексами  $i$ ,  $f$  и  $k$ . Вот что у нас есть (это может показаться необоснованным, но наберитесь терпения, я все объясню позже).

#### Листинг 10.20. 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 перенесенное состояние ( $c_t$ ), объединив  $i_t$ ,  $f_t$  и  $k_t$ .

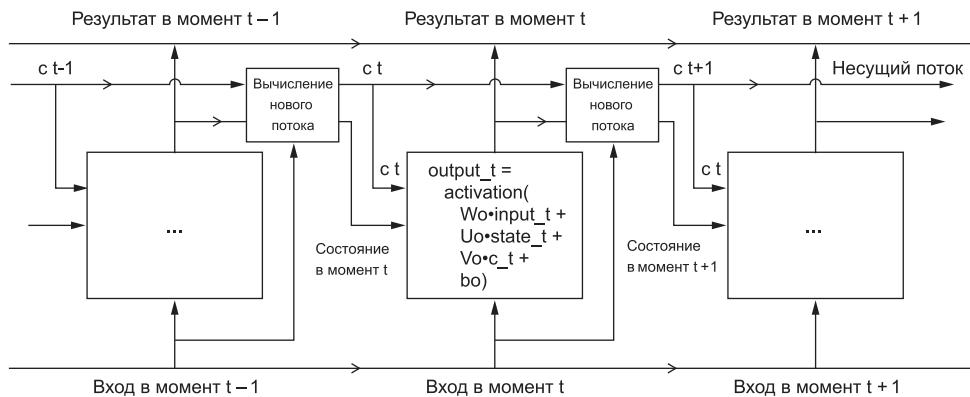
#### Листинг 10.21. Реализация архитектуры LSTM в псевдокоде (2/2)

```
c_t+1 = i_t * k_t + c_t * f_t
```

Добавим это в общую картину, как показано на рис. 10.10. 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 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. 10.10.** 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 особенностей 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.

## 10.4. УЛУЧШЕННЫЕ МЕТОДЫ ИСПОЛЬЗОВАНИЯ РЕКУРРЕНТНЫХ НЕЙРОННЫХ СЕТЕЙ

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. К концу раздела вы будете знать большую часть из того, что нужно знать об использовании рекуррентных сетей в Keras.

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);
- *дву направленные рекуррентные слои* — представляют одну и ту же информацию в рекуррентной сети разными способами, повышая точность и ослабляя проблемы, связанные с забыванием.

Мы используем эти приемы для совершенствования нашей рекуррентной сети предсказания температуры.

#### **10.4.1. Использование рекуррентного прореживания для борьбы с переобучением**

Вернемся к модели LSTM, которую мы создали в пункте 10.2.5, — нашей первой модели, сумевшей превзойти базовый уровень. Из кривых потерь на обучающих и проверочных данных (см. рис. 10.5) видно, что, несмотря на небольшое число параметров слоя, в модели быстро наступает эффект переобучения: потери на обучающих и проверочных данных 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 2016 году Ярин Гал в рамках своей докторской диссертации по байесовскому глубокому обучению<sup>1</sup> определил правильный способ применения прореживания к рекуррентным сетям: ко всем времененным 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-посто-

---

<sup>1</sup> Gal Y. Uncertainty in Deep Learning (PhD Thesis). October 13, 2016, <http://mng.bz/WBq1>.

янная 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.

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 LSTM и посмотрим, как это повлияет на переобучение.

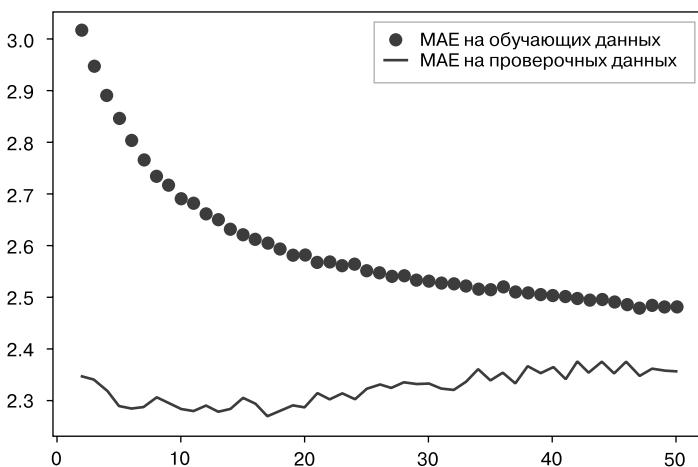
Прореживание позволяет в меньшей степени полагаться на размер сети для регуляризации, поэтому мы используем слой LSTM с вдвое большим количеством параметров, что, как я надеюсь, обеспечит большую выразительность (без прореживания такая сеть начала бы переобучаться практически сразу — попробуйте и убедитесь сами). Поскольку сети, регуляризованные с применением прореживания, всегда требуют больше времени для полной сходимости, обучим сеть за в два раза большее количество эпох.

**Листинг 10.22.** Обучение и оценка модели на основе LSTM с регуляризацией прореживанием

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)                                ← Для регуляризации слоя Dense добавим
outputs = layers.Dense(1)(x)                               также слой Dropout после LSTM
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                     epochs=50,
                     validation_data=val_dataset,
                     callbacks=callbacks)
```

Результаты показаны на рис. 10.11. Успех! Теперь эффект переобучения не наблюдается на протяжении первых 20 эпох. Мы достигли средней абсолютной ошибки 2,27 градуса на проверочных данных (на 7 % лучше, чем базовое решение без использования машинного обучения) и 2,45 градуса на контрольных данных (на 6,5 % лучше базового решения). Неплохо.



**Рис. 10.11.** Потери на этапах обучения и проверки модели на основе LSTM с прореживанием в задаче прогнозирования температуры по данным Jena

### О ПРОИЗВОДИТЕЛЬНОСТИ RNN

Рекуррентные модели с очень небольшим количеством параметров, такие как в этой главе, обычно работают значительно быстрее на простом многоядерном процессоре, чем на графическом, поскольку выполняют умножение небольших матриц, а цепочка умножений плохо поддается распараллеливанию из-за наличия цикла `for`. Но более крупные RNN могут значительно выиграть от выполнения на GPU.

При выполнении слоев LSTM и GRU на графическом процессоре с параметрами по умолчанию они будут использовать ядро cuDNN — высокооптимизированную реализацию базового алгоритма, созданную в NVIDIA (я упоминал ее в предыдущей главе). Однако ядра cuDNN неидеальны: они быстрые, но негибкие — поэтому, попытавшись сделать что-то, что не поддерживается ядром по умолчанию, вы столкнетесь с резким замедлением, которое заставит вас придерживаться того, что предлагает NVIDIA. Например, рекуррентное прореживание в LSTM и GRU не поддерживается ядрами cuDNN, поэтому его добавление в ваши слои заставит среду выполнения вернуться к рядовой реализации TensorFlow, которая обычно в 2–5 раз медленнее реализации на GPU (хотя имеет ту же вычислительную стоимость).

Чтобы ускорить работу рекуррентного слоя, когда нет возможности использовать cuDNN, попробуйте *развернуть* его. Разворачивание цикла `for` заключается в удалении инструкции цикла и простого повторения его тела  $N$  раз. В случае с RNN развертывание цикла `for` может помочь библиотеке

TensorFlow оптимизировать базовый граф вычислений. Правда при этом значительно увеличится потребление памяти вашей сетью RNN — в таком виде она будет пригодна только для обработки относительно небольших последовательностей (не более 100 шагов). Кроме того, поступить так можно, только если количество временных шагов в данных заранее известно (то есть если значение, передаваемое в параметре `shape` начального вызова `Input()`, не содержит `None`). Вот как это работает:

```
sequence_length  
не может быть None  
inputs = keras.Input(shape=(sequence_length, num_features)) ←  
x = layers.LSTM(32, recurrent_dropout=0.2, unroll=True)(inputs) ←  
Передайте unroll=True, чтобы  
разрешить развертывание
```

### 10.4.2. 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 и ее выразительной мощности. 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`.

В следующем примере мы попробуем создать стек из двух рекуррентных слоев с регуляризацией прореживанием. Для разнообразия вместо LSTM используем слои управляемых рекуррентных блоков (gated recurrent unit, GRU). Слой GRU очень похож на слой LSTM — это более простая и оптимизированная версия архитектуры LSTM. Он был представлен в 2014 году Чо с коллегами, когда

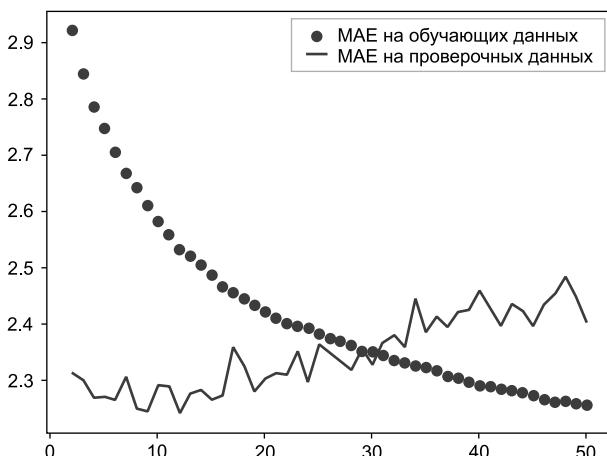
рекуррентные сети только начинали вновь вызывать интерес в крошечном исследовательском сообществе того времени<sup>1</sup>.

### Листинг 10.23. Обучение и оценка модели с несколькими слоями GRU и с регуляризацией прореживанияем

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_stacked_gru_dropout.keras",
        save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                     epochs=50,
                     validation_data=val_dataset,
                     callbacks=callbacks)
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

Результаты показаны на рис. 10.12. Мы достигли средней абсолютной ошибки 2,39 градуса (на 8,8 % лучше базового решения). Как видите, добавление слоя помогло немного улучшить результаты, хотя и незначительно. На данный момент вы можете наблюдать уменьшение отдачи от увеличения емкости сети.



**Рис. 10.12.** Потери на этапах обучения и проверки многослойной модели на основе GRU в задаче прогнозирования температуры по данным Jena

<sup>1</sup> Cho et al. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. 2014, <https://arxiv.org/abs/1409.1259>.

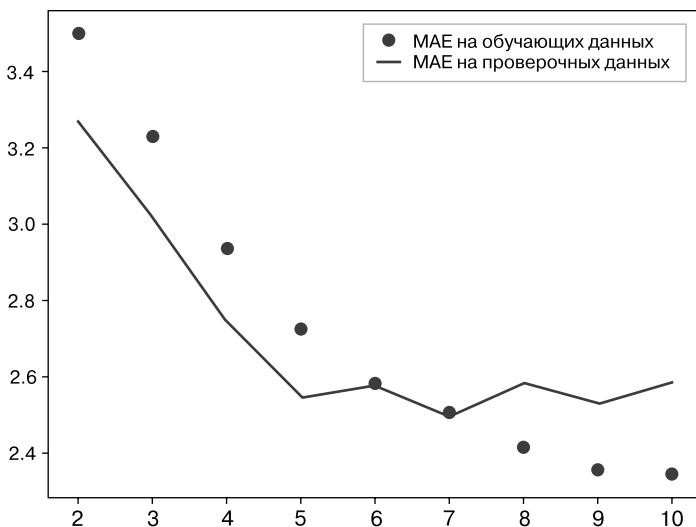
### 10.4.3. Использование двунаправленных рекуррентных нейронных сетей

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 обработка последовательностей в хронологическом порядке (от старых к новым) в данном разделе была выбрана 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 решение и посмотрим, что получится. Нужно лишь написать вариант генератора данных, обращающий входные последовательности (проще говоря, заменить последнюю строку инструкцией `yield samples[:, ::-1, :], targets`). Обучение той же модели на основе LSTM, которая использовалась в первом эксперименте данного раздела, дало результаты, представленные на рис. 10.13.

Сеть LSTM, обрабатывающая последовательности в обратном порядке, не достигает даже уровня базового решения — явное свидетельство того, что в данном случае хронологический порядок обработки имеет большое значение для успеха. Это вполне объяснимо: слой LSTM 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.

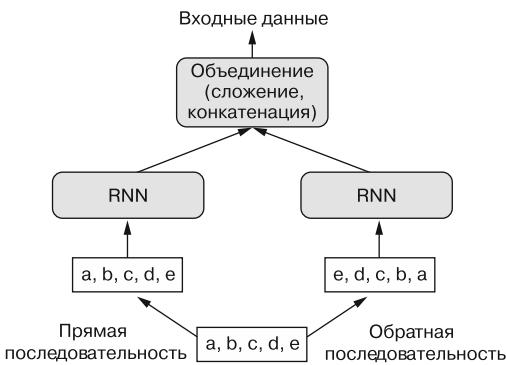


**Рис. 10.13.** Потери на этапах обучения и проверки модели на основе LSTM в задаче прогнозирования температуры по данным Jena с обучением на обращенных последовательностях

Следует отметить, что это не всегда верно для других задач, в том числе обработки естественных языков: очевидно, важность слова для понимания предложения обычно не зависит от его позиции в предложении. Обработка текстовых данных в обратном порядке дает результаты не хуже, чем обработка в прямом порядке, — человек может читать текст в обратном порядке и понимать его смысл (попробуйте!). Конечно, порядок слов важен для понимания языка, но *порядок их чтения* не имеет решающего значения.

Важно 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 13.

Двунаправленная рекуррентная сеть использует эту идею для улучшения качества обучения на упорядоченных данных. Она просматривает входную последовательность в обоих направлениях (рис. 10.14), получает потенциально более насыщенные представления и выделяет шаблоны, которые могли быть упущены однонаправленной версией.



**Рис. 10.14.** 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 прогнозирования температуры.

#### Листинг 10.24. Обучение и оценка двунаправленной модели LSTM

```

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                      epochs=10,
                      validation_data=val_dataset)
  
```

Качество этой модели по сравнению с обычным слоем `LSTM` ничуть не улучшилось. Легко понять почему: все прогностические способности исходят из половины сети, обрабатывающей данные в прямом хронологическом порядке, поскольку, как мы уже выяснили, качество половины, обрабатывающей данные в обратном порядке, в этой задаче сильно отстает (в данном случае недавнее прошлое имеет большее значение, чем отдаленное). В то же время наличие половины, обрабатывающей данные в обратном порядке, удваивает емкость сети, вследствие чего эффект переобучения наступает раньше.

Однако двунаправленные рекуррентные сети прекрасно подходят для обработки текстовых или любых других типов данных, где порядок имеет значение, но *используемый порядок* не так важен. Фактически в течение некоторого времени в 2016 году двунаправленные сети LSTM считались наиболее совершенным средством решения многих задач обработки естественного языка (до появления архитектуры Transformer, с которой вы познакомитесь в следующей главе).

#### 10.4.4. 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 потому наверняка не является оптимальным.
- Изменить скорость обучения с помощью оптимизатора RMSprop или попробовать другие оптимизаторы.
- Использовать несколько слоев Dense вместо одного как больший полно связанный регрессор поверх рекуррентных слоев.
- Улучшить входные данные: попробовать использовать более длинные или короткие последовательности, другую частоту дискретизации или выполнить процедуру проектирования признаков.

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.

#### 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.). Однако рынки имеют *совершенно иные статистические характеристики*, в отличие от таких природных явлений, как погода. Когда дело доходит до рынка, данные о прошлом служат плохой основой для предсказаний — невозможно двигаться вперед, глядя в зеркало заднего вида. С другой стороны, есть смысл применять машинное обучение к наборам данных, когда прошлое служит хорошим предсказателем будущего.

Всегда помните, что торговля — это, по сути, *информационное плутовство*: получение преимущества за счет использования данных или идей, отсутствующих у других участников рынка. Машинное обучение в задачах предсказания поведения рынка при наличии только общедоступных данных ведет в тупик из-за отсутствия информационного преимущества перед остальными. Скорее всего, вы просто потратите силы и время — и ничего не добьетесь.

По своему опыту могу сказать, что превышение уровня базового решения без обучения почти на 10 % — вероятно, лучшее, чего можно добиться с этим набором данных. Это не самый лучший результат, но он вполне объясним: погода в ближайшем будущем весьма предсказуема, если имеется доступ к данным, полученным из широкой сети метеостанций, но ее трудно предсказать при наличии измерений, сделанных только на одной метеостанции. Изменение погоды в месте, где вы находитесь, зависит также от текущих погодных условий в соседних районах.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Как мы узнали еще в главе 5, 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 (особенно это касается временных последовательностей), лучше всего подходят *рекуррентные сети* — они с легкостью превосходят модели, которые сначала снижают размерность исходных данных. В Keras доступны два основных слоя рекуррентных сетей: `LSTM` и `GRU`.
- 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 `recurrent_dropout`.
- Комбинации из нескольких рекуррентных слоев обеспечивают большую репрезентативность, чем один слой. Они также являются намного более затратными с точки зрения вычислений, и поэтому их применение не всегда оправданно. Они позволяют повысить качество решения сложных задач (таких как машинный перевод), но не всегда подходят для небольших и простых задач.

# 11

## *Глубокое обучение для текста*

### **В этой главе**

- ✓ Подготовка текстовых данных для приложений машинного обучения.
- ✓ Методики «мешок слов» и другие для обработки текста и последовательностей.
- ✓ Архитектура Transformer.
- ✓ Обучение типа «последовательность в последовательность».

### **11.1. ОБРАБОТКА ЕСТЕСТВЕННЫХ ЯЗЫКОВ**

В информатике языки человеческого общения, такие как английский или китайский, называются естественными языками, чтобы отличить их от машинных — например, ассемблера, LISP или XML. Все машинные языки были *созданы искусственно*: у истоков каждого стоял человек — инженер, определивший набор формальных правил, устанавливающих, какие утверждения можно делать и что они означают. Правила были первоосновой — соответственно, начать использовать язык можно было только после того, как набор правил был окончательно сформулирован. С человеческим языком все иначе: сначала в обиходе появился язык, и лишь потом стали формироваться правила, его описывающие. Естественные языки формировались эволюционно, как и биологические организмы, — именно это делает их естественными. Их правила и грамматика кристаллизовались постфактум и, несмотря на свою формальность, сегодня часто

игнорируются или нарушаются пользователями. В результате машинные языки хорошо структурированы, имеют строгие и точные синтаксические правила и определяют понятия из фиксированного словаря; естественные же языки запутанны, неоднозначны, хаотичны и постоянно меняются.

Создание алгоритмов, способных понимать естественный язык, — сложная задача. Язык и, в частности, текст лежат в основе нашего общения и культурного производства. Информационное наполнение интернета по большей части текстовое. Язык — наше средство хранения знаний. Мы даже мыслим на определенном языке. Однако машинам способность понимать естественный язык долгое время была не под силу. Когда-то многие наивно полагали, что можно просто записать «набор правил естественного языка» подобным образом, как записан набор правил LISP. Поэтому ранние попытки создания систем обработки естественного языка (Natural Language Processing, NLP) предпринимались с позиции «прикладной лингвистики». Инженеры и лингвисты вручную создавали комплексные наборы правил для реализации базового машинного перевода или простых чат-ботов — например, знаменитая программа ELIZA, написанная в 1960-х годах, для поддержания простейшей беседы использовала прием сопоставления с образцом. Но естественный язык — сложная штука: он с большим трудом поддается формализации. После нескольких десятилетий усилий возможности этих систем так и остались на низком уровне.

Собственноручное создание правил оставалось доминирующим подходом вплоть до 1990-х годов. Но начиная с конца 1980-х появление более быстрых компьютеров и доступность больших объемов данных сделали возможной более удачную альтернативу. Поймав себя на попытке построить систему, организованную в виде огромной груды специальных правил, вы, как умный инженер, вероятно, зададите такие вопросы: «Можно ли автоматизировать процесс поиска этих правил, используя корпус данных? Можно ли искать правила в некотором их пространстве, вместо того чтобы придумывать их?» И вот так мы переходим к машинному обучению. Подходы к обработке естественного языка на основе машинного обучения начали появляться в конце 1980-х годов. Самые ранние из них опирались на деревья решений — их целью было буквально автоматизировать разработку правил вида «если/то/иначе» по аналогии с предыдущими системами. Затем, начиная с логистической регрессии, стали набирать обороты статистические подходы. Со временем верх взяли модели с обучаемыми параметрами — и лингвистика стала рассматриваться больше как помеха, чем как полезный инструмент. Фредерик Елинек, один из первых исследователей в области распознавания речи, в 1990-х годах пошутил: «Каждый раз, когда я увольняю лингвиста, качество модели распознавания речи повышается».

Именно в этом суть современной обработки естественного языка. Использование машинного обучения и больших наборов данных дает компьютерам

возможность не *понимать* язык, что является более высокой целью, а принимать фрагмент данных на естественном языке и возвращать что-то полезное. Например, определять:

- тему текста (задача классификации текста);
- наличие в тексте оскорбительных слов (фильтрация содержимого);
- эмоциональную окраску текста (анализ эмоциональной окраски);
- следующее слово в незаконченном предложении (моделирование языка);
- то же самое выражение на немецком языке (перевод);
- формулировку основной идеи в одном абзаце (обобщение) и т. д.

Само собой, во время изучения главы держите в памяти то, что обучаемые нами модели не будут обладать человеческим пониманием языка — они будут искать статистические закономерности во входных данных, чего вполне достаточно, чтобы хорошо справиться со многими простыми задачами. По сути, как компьютерное зрение распознает образы, применяемые к пикселям, так и обработка естественного языка — это распознавание образов, применяемое к словам, предложениям и абзацам.

Инструменты обработки естественного языка (деревья решений, логистическая регрессия) медленно развивались с 1990-х до начала 2010-х годов. Основные исследования были сосредоточены на проектировании признаков. Когда я выиграл свое первое состязание по обработке естественного языка на Kaggle в 2013 году, моя модель, как вы наверняка догадались, была основана на деревьях решений и логистической регрессии. Однако примерно в 2014–2015 годах ситуация начала меняться. Несколько исследователей стали изучать возможность применения для обработки естественного языка рекуррентных нейронных сетей, в частности не замеченного до того времени LSTM — алгоритма обработки последовательностей, появившегося в конце 1990-х годов.

В начале 2015 года компания Keras представила первую простую реализацию LSTM с открытым исходным кодом — как раз в начале новой волны интереса к рекуррентным нейронным сетям (до этого существовал только «исследовательский код», который невозможно было использовать повторно). С 2015 по 2017 год рекуррентные нейронные сети заняли доминирующие позиции в бурно развивающейся области обработки естественного языка. Двунаправленные модели LSTM, в частности, задали современный уровень во многих важных задачах, от обобщения до ответов на вопросы и машинного перевода.

Наконец, примерно в 2017–2018 годах на смену рекуррентным сетям пришла новая архитектура Transformer, с которой вы познакомитесь во второй половине этой главы. Она позволила за короткий период времени добиться значительного прогресса в рассматриваемых вопросах, и в настоящее время большинство систем обработки естественного языка основаны на ней.

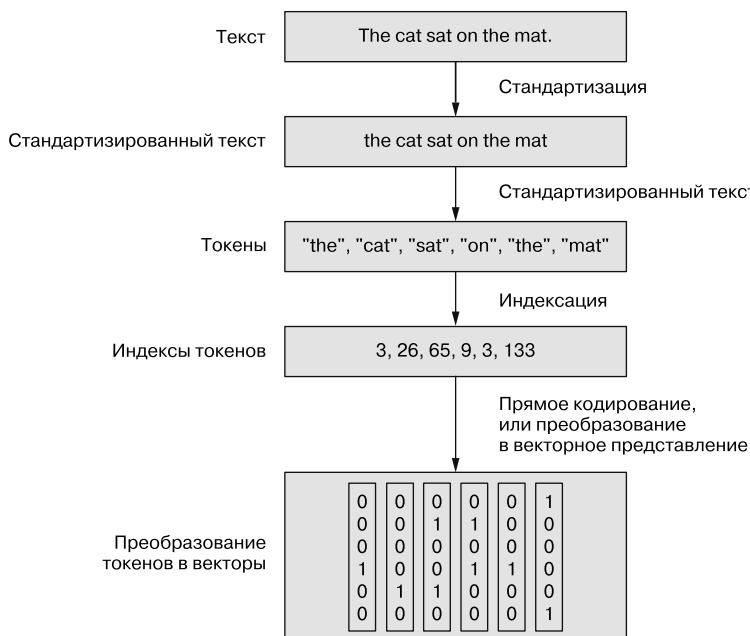
А теперь рассмотрим детали. Далее мы пройдем весь путь реализации моделей машинного перевода с самого начала с помощью Transformer.

## 11.2. ПОДГОТОВКА ТЕКСТОВЫХ ДАННЫХ

Модели глубокого обучения, как дифференцируемые функции, могут обрабатывать только числовые тензоры: они не принимают на входе необработанный текст. Процесс преобразования текста в числовые тензоры называется *векторизацией*. Процессы векторизации текста бывают разных видов и форм, но все протекают по одному шаблону (рис. 11.1):

- сначала текст стандартизируется, чтобы упростить его обработку: например, все символы преобразуются в нижний регистр или из текста удаляются знаки препинания;
- затем текст разбивается на единицы (называемые *токенами*) — символы, слова или группы слов; данный процесс называется *токенизацией*;
- после этого каждый токен преобразуется в числовой вектор, обычно путем индексации всех токенов, присутствующих в данных.

Давайте рассмотрим каждый из этих шагов поближе.



**Рис. 11.1.** Порядок преобразования простого текста в векторы

### 11.2.1. Стандартизация текста

Рассмотрим два текста:

- *sunset came. i was staring at the Mexico sky. Isnt nature splendid??;*
- *Sunset came; I stared at the México sky. Isn't nature splendid?.*

Они очень похожи, а по смыслу так вообще идентичны: «Солнце клонится к закату. Я смотрел на мексиканское небо. Разве природа не прекрасна?» Однако, если преобразовать их в последовательности байтов, получатся очень разные представления: буквы *i* и *I* — это два разных символа, *Mexico* и *México* — два разных слова, *isnt* не равно *isn't* и т. д. Модель машинного обучения не знает, что *i* и *I* — это одна и та же буква, что *é* — это *e* с ударением или что *staring* и *stared* — две формы одного и того же глагола.

Стандартизация текста — это базовая форма проектирования признаков, целью которой является стирание различий между представлениями, с которыми ваша модель должна иметь дело. Это касается не только машинного обучения — при создании поисковой системы вам пришлось бы делать то же самое.

Одна из самых простых и распространенных схем стандартизации — «преобразовать в нижний регистр и удалить знаки препинания». После ее применения наши два текста превратятся в:

- *sunset came i was staring at the mexico sky isnt nature splendid;*
- *sunset came i stared at the méxico sky isnt nature splendid.*

Тексты стали еще ближе друг к другу. Следующее распространенное преобразование — приведение специальных символов в стандартную форму: например замена *é* на *e*, *æ* на *ae* и т. д. После этого наш токен *méxico* превратится в *mexico*.

Есть еще более продвиннутое преобразование, которое, впрочем, редко используется в контексте машинного обучения. Речь идет о *стемминге* — выделении корня (скажем, из различных форм спряжения глагола) для преобразования в единую форму: *caught* («пойман») и *been catching* («был пойман») в *[catch]* («поймать») или *cats* («кошки») в *[cat]* («кошка»). После стемминга *was staring* и *stared* превращаются в *[stare]* — и два похожих текста наконец получают одинаковое представление:

- *sunset came i [stare] at the mexico sky isnt nature splendid.*

Благодаря этим методам стандартизации модели потребуется меньше данных для обучения и она будет лучше их обобщать. Ей не понадобятся многочисленные примеры типа *Sunset* и *sunset*, чтобы понять, что они означают одно и то же; и она сможет увидеть смысл слова *México*, даже если в обучающем наборе будет

присутствовать только форма *mexico*. Конечно, стандартизация может также стереть часть информации, поэтому всегда помните о контексте: например, если вы пишете модель, извлекающую вопросы из интервью, она обязательно должна обрабатывать вопросительный знак (?) как отдельный токен, а не отбрасывать его, ведь для данной задачи это полезный сигнал.

### **11.2.2. Деление текста на единицы (токенизация)**

После стандартизации текста его нужно разбить на единицы (токены) для векторизации. Этот шаг называется *токенизацией*. Токенизацию можно выполнить тремя разными способами:

- *токенизацией на уровне слов* — токенами являются подстроки, разделенные пробелами (или знаками препинания). Один из вариантов — дальнейшее деление слов на подслова, когда это применимо: например, *staring* на *star+ing* или *called* на *call+ed*;
- *токенизацией на N-граммы* — токенами являются группы из  $N$  слов, следующих друг за другом. Например, *the cat* или *he was* можно превратить в 2-граммы (их также называют биграммами);
- *токенизацией на уровне символов* — каждый символ является отдельным токеном (на практике данная схема используется редко и обычно применяется только в специализированных контекстах, таких как генерация текста или распознавание речи).

В целом обычно используется токенизация на уровне слов или  $N$ -грамм. Существует два типа моделей обработки текста: которые учитывают порядок следования слов (их называют моделями последовательностей) и которые обрабатывают входные слова как простое неупорядоченное множество (их называют *мешками слов*). При создании последовательных моделей вы будете использовать токенизацию на уровне слов, а при создании модели мешка слов — токенизацию на  $N$ -граммы.  $N$ -граммы позволяют искусственно внедрить в модель минимальный объем информации о порядке слов. В этой главе вы познакомитесь с обоими типами моделей и особенностями их использования.

#### **N-ГРАММЫ И МЕШКИ СЛОВ**

*N*-граммы слов — это группы из  $N$  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:

```
{"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:

```
{"the", "the cat", "cat", "cat sat", "the cat sat",
"sat", "sat on", "on", "cat sat on", "on the", "the",
"sat on the", "the mat", "mat", "on the mat"}
```

Подобные наборы называют *мешком bigрамм* 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* (или *мешком N-грамм*).

Поскольку мешок слов не сохраняет порядок следования токенов (сгенерированный набор токенов интерпретируется как множество, а не как последовательность, и не поддерживает общей структуры предложений), данный метод обычно используется в поверхностных моделях обработки естественного языка и крайне редко — в моделях глубокого обучения. Извлечение *N-грамм* — еще одна форма проектирования признаков, но в глубоком обучении этот ломкий и ограниченный метод заменяют проектированием иерархических признаков. Одномерные сверточные и рекуррентные нейронные сети, а также архитектура Transformer способны получать представления для групп слов и символов без явного определения таких групп, просто просматривая последовательности слов или символов.

### 11.2.3. Индексирование словаря

После разбивки текста на токены их нужно закодировать в числовое представление. Это можно сделать простым хешированием каждого токена в фиксированный двоичный вектор, но на практике чаще используется другой способ — построение индекса (словаря) всех терминов, найденных в обучающих данных, и назначение уникального целого числа каждому его элементу.

Например так:

```
vocabulary = {}
for text in dataset:
    text = standardize(text)
    tokens = tokenize(text)
    for token in tokens:
        if token not in vocabulary:
            vocabulary[token] = len(vocabulary)
```

После этого целые числа можно преобразовать в векторное представление, которое может быть обработано нейронной сетью:

```
def one_hot_encode_token(token):
    vector = np.zeros((len(vocabulary),))
    token_index = vocabulary[token]
    vector[token_index] = 1
    return vector
```

Обратите внимание, что на этом этапе словарный запас обычно ограничивают 20 или 30 тысячами самых распространенных слов, найденных в обучающих данных. Любой набор текстовых данных, как правило, содержит чрезвычайно большое количество уникальных терминов, большинство из которых появляются только один или два раза, — индексирование таких редких терминов привело бы к чрезмерному увеличению пространства признаков, в котором большинство признаков почти не несют информации.

Помните, как вы обучали свои первые модели глубокого обучения на наборе данных IMDB в главах 4 и 5? Данные из `keras.datasets.imdb`, которые вы использовали, уже были предварительно обработаны и преобразованы в последовательности целых чисел, где каждое целое число означало определенное слово. Тогда мы использовали параметр `num_words=10000`, чтобы ограничить словарь 10 000 самых распространенных слов, найденных в обучающих данных.

Теперь отметим существенную деталь, которую нельзя упускать из виду: новый разыскиваемый в словаре токен может там отсутствовать. Возможно, в обучающих данных не найдется слово `cherimoya` (или вы сами можете исключить его из индекса, потому что оно слишком редкое), поэтому выполнение инструкции `token_index = word["cherimoya"]` может привести к исключению `KeyError`. Чтобы этого избежать, нужно предусмотреть индекс для токенов «вне словаря», который будет назначаться всем токенам, отсутствующим в словаре. Обычно это индекс 1: в таком случае вы должны определять индекс инструкции `token_index = word.get(token, 1)`. При декодировании последовательности целых чисел обратно в слова данное число можно заменять на что-то вроде `[UNK]` («несловарный токен»).

Вы можете спросить: «Почему именно 1, а не 0?» Дело в том, что число 0 уже занято *токеном маски* (индекс 0). Это специальный токен, такой же как несловарный токен (индекс 1). Несловарный токен означает «здесь было слово, которое не удалось распознать», а токен маски говорит: «Игнорируйте меня, я не слово». Его можно использовать, в частности, для дополнения последовательностей в данных: так как пакеты с данными должны быть непрерывными, всем последовательностям в пакете нужно задать одинаковую длину, поэтому более короткие последовательности дополняются до длины самой длинной последовательности. Если потребуется создать пакет с последовательностями `[5, 7, 124, 4, 89]` и `[8, 34, 21]`, он должен выглядеть так:

```
[[5, 7, 124, 4, 89]
 [8, 34, 21, 0, 0]]
```

Именно таким образом нулями были дополнены пакеты последовательностей целых чисел, образованных из набора данных IMDB, с которыми вы работали в главах 4 и 5.

### 11.2.4. Использование слоя TextVectorization

Все перечисленные до сих пор шаги легко реализовать на чистом Python. Например, вы могли бы написать такой код:

```
import string

class Vectorizer:
    def standardize(self, text):
        text = text.lower()
        return "".join(char for char in text
                      if char not in string.punctuation)

    def tokenize(self, text):
        text = self.standardize(text)
        return text.split()

    def make_vocabulary(self, dataset):
        self.vocabulary = {"": 0, "[UNK)": 1}
        for text in dataset:
            text = self.standardize(text)
            tokens = self.tokenize(text)
            for token in tokens:
                if token not in self.vocabulary:
                    self.vocabulary[token] = len(self.vocabulary)
        self.inverse_vocabulary = dict(
            (v, k) for k, v in self.vocabulary.items())

    def encode(self, text):
        text = self.standardize(text)
        tokens = self.tokenize(text)
        return [self.vocabulary.get(token, 1) for token in tokens]

    def decode(self, int_sequence):
        return " ".join(
            self.inverse_vocabulary.get(i, "[UNK]") for i in int_sequence)

vectorizer = Vectorizer()
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",   | Хайку японского
    "A poppy blooms.",       | поэта Хокуси
]
vectorizer.make_vocabulary(dataset)
```

И с его помощью подготовить свои данные:

```
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = vectorizer.encode(test_sentence)
>>> print(encoded_sentence)
[2, 3, 5, 7, 1, 5, 6]
>>> decoded_sentence = vectorizer.decode(encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

Однако это не лучшее решение. На практике обычно используется слой `TextVectorization` из библиотеки Keras. Он работает быстро и эффективно, и его можно добавить непосредственно в конвейер `tf.data` или в модель Keras.

Вот как создается слой `TextVectorization`:

```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization(
    output_mode="int", ←
) ← Слой настраивается для возврата последовательностей слов,
    предоставленных в виде целочисленных индексов. Доступно несколько
    других режимов вывода, с которыми вы вскоре познакомитесь
```

По умолчанию слой `TextVectorization` использует режим стандартизации «преобразование символов в нижний регистр и удаление знаков пунктуации» и токенизации «разделение по пробелам». Но, что особенно важно, вы можете передать свои функции для стандартизации и токенизации — как видим, слой достаточно гибок и подходит для любых вариантов использования. Помните, что эти функции должны работать с тензорами `tf.string`, а не с обычными строками Python! Для примера рассмотрим код ниже, представляющий поведение слоя по умолчанию:

```
import re
import string
import tensorflow as tf
def custom_standardization_fn(string_tensor):
    lowercase_string = tf.strings.lower(string_tensor) ←
    ← Преобразовать строки
    ← в нижний регистр
    return tf.strings.regex_replace(
        lowercase_string, f"[{re.escape(string.punctuation)}]", "") ←
    ← Заменить знаки
    ← пунктуации пустыми
    ← строками
def custom_split_fn(string_tensor):
    return tf.strings.split(string_tensor) ←
    ← Разбить строки
    ← по пробелам
text_vectorization = TextVectorization(
    output_mode="int",
    standardize=custom_standardization_fn,
    split=custom_split_fn,
)
```

Чтобы проиндексировать словарь корпуса текста, достаточно вызвать метод `adapt()` слоя с объектом `Dataset`, возвращающим строки, или просто со списком строк Python:

```
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
text_vectorization.adapt(dataset)
```

Обратите внимание на то, что созданный словарь можно получить вызовом `get_vocabulary()` — это может пригодиться в случае, если текст, представленный в виде последовательности целых чисел, понадобится преобразовать обратно в слова. Первые две записи в словаре — это токен маски (индекс 0) и несловарный токен (индекс 1). Записи в словаре отсортированы по частоте, поэтому в реальном наборе данных самые распространенные слова, такие как артикли *the* или *a*, будут находиться на первых местах.

### Листинг 11.1. Вывод содержимого словаря

```
>>> text_vectorization.get_vocabulary()
[ "", "[UNK]", "erase", "write", ... ]
```

Для демонстрации попробуем закодировать, а затем декодировать предложение:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7 3 5 9 1 5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

#### СЛОЙ TEXTVECTORIZATION КАК ЧАСТЬ МОДЕЛИ ИЛИ КОНВЕЙЕРА TF.DATA

Важно отметить, что основной операцией в `TextVectorization` является операция поиска по словарю, которую нельзя выполнить на GPU или TPU — только на CPU. Поэтому при обучении модели на GPU слой `TextVectorization` сначала выполнится на CPU — и только потом его выходные данные будут отправлены в GPU. Это имеет важные последствия для производительности.

Есть два способа использования слоя `TextVectorization`. Первый — поместить его в конвейер `tf.data`, например — так:

```
int_sequence_dataset = string_dataset.map( ←
    text_vectorization,
    num_parallel_calls=4) ←
                                Аргумент num_parallel_calls используется для параллельного
                                выполнения map() на нескольких ядрах CPU
```

string\_dataset может быть набором  
данных, возвращающим строковые  
тензоры

Второй способ — включить слой в состав модели (в конце концов, это слой Keras), как показано ниже:

```
Определить форму строковых входных данных
text_input = keras.Input(shape=(), dtype="string") ←
vectorized_text = text_vectorization(text_input) ←
Применить
embedded_input = keras.layers.Embedding(...)(vectorized_text)
output = ...
model = keras.Model(text_input, output)
```

Далее можно добавлять новые слои, как это делается  
при создании моделей с помощью функционального API

Эти два способа имеют важное различие. Если этап векторизации является частью модели, он будет выполняться синхронно с остальной частью модели, то есть на каждом этапе обучения остальная часть модели (выполняющаяся на GPU) будет ждать, пока слой TextVectorization (на CPU) подготовит данные. При размещении же слоя в конвейере tf.data появляется возможность проводить предварительную обработку данных на CPU асинхронно: пока модель на GPU обрабатывает один пакет векторизованных данных, CPU выполняет векторизацию следующего пакета исходных строк.

Поэтому, если модель обучается на GPU или TPU, вероятно, лучше выбрать первый способ, чтобы получить максимальную производительность. Так мы и поступим во всех практических примерах далее в этой главе. Однако при обучении на CPU можно использовать и синхронную обработку: вы в любом случае получите 100%-ную загрузку ядер процессора, независимо от выбранного способа.

Кроме того, если предполагается экспортовать модель в производственное окружение, то предпочтительнее выглядит модель, принимающая исходные строки (как в примере, демонстрирующем второй способ). В противном случае придется повторно реализовать стандартизацию и токенизацию текста в производственном окружении (возможно, на JavaScript?); к тому же увеличивается риск небольших расхождений в процессе предварительной обработки, которые могут ухудшить точность модели. К счастью, слой TextVectorization позволяет вам включить предварительную обработку текста прямо в модель, упрощая ее развертывание, даже если первоначально слой использовался как часть конвейера tf.data. Во врезке «Экспорт модели, обрабатывающей исходные строки» вы узнаете, как экспортовать обученную модель, предназначенную только для прогнозирования, которая включает предварительную обработку текста.

Теперь вы знаете о предварительной обработке текста все, что нужно. Давайте перейдем к этапу моделирования.

## 11.3. ДВА ПОДХОДА К ПРЕДСТАВЛЕНИЮ ГРУПП СЛОВ: МНОЖЕСТВА И ПОСЛЕДОВАТЕЛЬНОСТИ

Представление *отдельных слов* в моделях машинного обучения почти не вызывает сомнений: это категориальные признаки (значения из предопределенного набора), и мы знаем, как с ними обращаться. Они должны быть представлены как измерения в пространстве признаков или как векторы категорий (в данном случае векторы слов). Более серьезный вопрос касается способа кодирования *переплетения слов в предложениях* — иными словами, порядка их следования.

Проблема порядка следования в естественном языке весьма интересна: в отличие от интервалов во временных последовательностях, слова в предложениях не имеют натурального канонического порядка. В разных языках похожие слова упорядочиваются по-разному. Например, структура предложений в английском и японском языках сильно различается. Даже в одном языке ту же самую мысль можно выразить по-разному, просто переставляя выражения местами. Более того, даже перемешав в случайном порядке слова в коротком предложении, вы все равно с большой долей вероятности сможете понять его смысл, хотя нередко в подобных ситуациях могут возникнуть существенные разнотечения. Порядок, конечно же, важен, но его связь с сутью предложения не особенно тесная.

Представление порядка слов — ключевой вопрос, из которого вытекают различные виды архитектур моделей. Самое простое, что можно сделать, — отбросить порядок и рассматривать текст как неупорядоченный набор слов. Такой подход дает нам *модели мешка слов*. С другой стороны, можно решить, что слова должны обрабатываться строго в том порядке, в каком они следуют в предложениях, по одному, как интервалы временной последовательности, и для этого можно использовать рекуррентные модели, представленные в предыдущей главе. Наконец, возможен гибридный подход: архитектура Transformer технически не зависит от порядка, но учитывает информацию о положении слов, что позволяет ей одновременно рассматривать разные части предложения (в отличие от RNN), принимая во внимание также их порядок. Поскольку RNN и Transformer учитывают порядок следования слов, их называют *моделями последовательностей*.

Так сложилось, что в самых ранних случаях применения машинного обучения для обработки естественного языка использовались модели мешка слов. Интерес к моделям последовательностей проснулся только в 2015 году, с возрождением рекуррентных нейронных сетей. В настоящее время оба подхода остаются актуальными. Давайте посмотрим, как они работают и в каких случаях их можно использовать.

Я продемонстрирую каждый из них на известной задаче классификации текста: анализе отзывов в наборе данных IMDB. В главах 4 и 5 мы уже пробовали работать с предварительно векторизованной версией набора данных IMDB; теперь обрабатываем исходные текстовые данные IMDB самостоятельно, как если бы перед нами стояла совершенно новая задача классификации текста.

### **11.3.1. Подготовка данных IMDB с отзывами к фильмам**

Сначала загрузите набор данных со страницы Эндрю Мааса на сайте Стэнфордского университета и распакуйте его:

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz  
!tar -xf aclImdb_v1.tar.gz
```

У вас должен появиться каталог aclImdb со следующей структурой:

```
aclImdb/  
...train/  
.....pos/  
.....neg/  
...test/  
.....pos/  
.....neg/
```

Каталог `train/pos/`, например, содержит 12 500 текстовых файлов с положительными отзывами о фильмах, которые будут использоваться в качестве обучающих данных. Отрицательные отзывы находятся в каталогах `neg`. Всего имеется 25 000 текстовых файлов для обучения и еще 25 000 для контроля.

Также имеется подкаталог `train/unsup`, который нам не понадобится. Его можно удалить:

```
!rm -r aclImdb/train/unsup
```

Давайте заглянем в некоторые из этих файлов. С чем бы вы ни работали — с текстовыми данными или с изображениями, — всегда проверяйте, как выглядят

ваши данные, прежде чем погрузиться в их моделирование. Это укрепит ваши интуитивные представления о том, что в действительности делает модель:

```
!cat aclImdb/train/pos/4077_10.txt
```

Далее подготовим проверочный набор, отобрав 20 % обучающих файлов и поместив их в новый каталог `aclImdb/val`:

```
import os, pathlib, shutil, random

base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)
```

Перемешать список файлов из обучающего набора с помощью определенного начального значения для генератора случайных чисел, чтобы обеспечить выбор одних и тех же файлов в проверочный набор

Выбрать 20 % обучающих файлов для использования в качестве проверочных данных

Переместить файлы в каталоги `aclImdb/val/neg` и `aclImdb/val/pos`

Вспомните, как в главе 8 мы использовали утилиту `image_dataset_from_directory` для создания пакетного набора данных `Dataset` с изображениями и соответствующими им метками, которые определяются по структуре каталогов. То же самое можно проделать с текстовыми файлами. Создадим три объекта `Dataset` для обучающих, проверочных и контрольных данных:

```
from tensorflow import keras
batch_size = 32

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size) ←
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
```

Эта инструкция должна вывести сообщение: Found 20 000 files belonging to 2 classes (Найдено 20 000 файлов, принадлежащих двум классам); если вы увидите сообщение Found 70 000 files belonging to 3 classes (Найдено 70 000 файлов, принадлежащих трем классам), значит, вы забыли удалить каталог `aclImdb/train/unsup`

Эти наборы данных возвращают входные данные в форме тензоров `tf.string` с исходными строками и целочисленных тензоров с целевыми значениями 0 и 1.

### Листинг 11.2. Вывод форм и типов содержимого первого пакета

```
>>> for inputs, targets in train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
```

```
>>> print("targets.shape:", targets.shape)
>>> print("targets.dtype:", targets.dtype)
>>> print("inputs[0]:", inputs[0])
>>> print("targets[0]:", targets[0])
>>> break
inputs.shape: (32,)
inputs.dtype: <dtype: "string">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
inputs[0]: tf.Tensor(b"This string contains the movie review.", shape=(),
                     dtype=string)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

Наборы данных готовы. Теперь попробуем извлечь какие-нибудь знания из этих данных.

### 11.3.2. Обработка наборов данных: мешки слов

Самый простой способ закодировать текст для последующей обработки моделью машинного обучения — представить его как неупорядоченное множество («мешок») токенов. При этом можно рассматривать слова по отдельности (как униграммы) или попытаться сохранить часть информации о порядке слов, рассматривая их группами из следующих друг за другом токенов (как  $N$ -граммы).

#### Бинарное кодирование отдельных слов (униграммы)

Если использовать набор отдельных слов, предложение *The cat sat on the mat* («Кошка села на коврик») примет следующую форму:

```
{"cat", "mat", "on", "sat", "the"}
```

Основное преимущество такого способа кодирования — возможность представить весь текст в виде одного вектора, каждый элемент которого является индикатором присутствия одного слова. Например, при бинарном (multi hot) кодировании текст преобразуется в вектор с количеством измерений, равным количеству слов в словаре, — с нулями почти везде и единицами в тех измерениях, которые представляют присутствующие в тексте слова. Именно с таким представлением мы работали в главах 4 и 5. Давайте попробуем использовать его в нашей задаче.

Сначала обработаем наши наборы текстовых данных с помощью слоя `TextVectorization`, чтобы получить бинарные векторы слов, полученные применением федеративного кодирования. Наш слой будет рассматривать слова по одному (то есть как униграммы).

**Листинг 11.3.** Предварительная обработка наборов данных с помощью слоя TextVectorization

```

Ограничите размер словаря
20 000 наиболее часто встречающихся слов.
Иначе придется индексировать каждое
слово в обучающих данных, десятки
тысяч из которых могут встречаться
только один-два раза и, соответственно,
не являются информативными. Обычно
20 000 — это оптимальный размер словаря
для классификации текстов

text_vectorization = TextVectorization(
    max_tokens=20000,
    output_mode="multi_hot",
)
text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)

binary_1gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_1gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_1gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)

```

Посмотрим, что получилось в результате.

**Листинг 11.4.** Исследование содержимого набора данных с бинарными униграммами

```

>>> for inputs, targets in binary_1gram_train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32, 20000)           ← Входы — пакеты
inputs.dtype: <dtype: "float32">   ← 20 000-мерных векторов
targets.shape: (32,)                ← Эти векторы состоят
targets.dtype: <dtype: "int32">      из нулей и единиц
inputs[0]: tf.Tensor([1. 1. 1. ... 0. 0. 0.], shape=(20000,), dtype=float32) ←
targets[0]: tf.Tensor(1, shape=(), dtype=int32)

```

Теперь напишем функцию, конструирующую модель, которую мы будем использовать во всех экспериментах в этом разделе.

**Листинг 11.5.** Функция конструирования модели

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(max_tokens=20000, hidden_dim=16):
    inputs = keras.Input(shape=(max_tokens,))
    x = layers.Dense(hidden_dim, activation="relu")(inputs)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model
```

Наконец, обучим и проверим модель на контрольных данных.

**Листинг 11.6.** Обучение модели на бинарных униграммах и ее проверка

```
model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("binary_1gram.keras",
                                    save_best_only=True)
]
model.fit(binary_1gram_train_ds.cache(),
          validation_data=binary_1gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("binary_1gram.keras")
print(f"Test acc: {model.evaluate(binary_1gram_test_ds)[1]:.3f}")
```

**Метод cache()** наборов данных применяется для кеширования их в памяти. Благодаря этому предварительная обработка будет выполняться только один раз в первой эпохе, а в остальные эпохи будут использоваться уже обработанные данные. Этим приемом можно пользоваться, только если данных не слишком много и они могут уместиться в памяти

Модель достигла точности 89,2 % на контрольных данных — неплохо! Обратите внимание, что в этом случае набор данных сбалансирован (положительных образцов столько же, сколько и отрицательных), поэтому «базовый уровень», которого можно достичь без обучения реальной модели, составляет 50 %. Между тем наилучшая оценка точности на контрольных данных, которую можно получить на этом наборе без использования внешних данных, близка к 95 %.

**Бинарное кодирование пар слов (биграмм)**

Как вы наверняка понимаете, отказ от учета порядка слов сильно влияет на точность, потому что даже атомарные понятия часто выражаются несколькими словами: пара слов «Соединенные Штаты» передает понятие, совершенно отличное от взятых по отдельности значений слов «штаты» и «соединенные».

По этой причине в конечном счете приходится вновь вводить в представление мешка слов информацию о локальном порядке, объединяя слова в  $N$ -граммы (чаще всего в биграммы).

В случае с биграммами наше предложение про кошку превращается в следующий набор данных:

```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

Слой `TextVectorization` можно настроить так, чтобы он возвращал произвольные  $N$ -граммы: биграммы, триграмммы и т. д. Для этого достаточно передать аргумент `ngrams=N`, как показано в следующем листинге.

**Листинг 11.7.** Настройка слоя `TextVectorization` для получения биграмм

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="multi_hot",
)
```

Посмотрим, как изменится качество модели после обучения на мешке бинарных биграмм.

**Листинг 11.8.** Обучение модели на бинарных биграммах и ее проверка

```
text_vectorization.adapt(text_only_train_ds)
binary_2gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_2gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_2gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)

model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("binary_2gram.keras",
                                    save_best_only=True)
]
model.fit(binary_2gram_train_ds.cache(),
          validation_data=binary_2gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("binary_2gram.keras")
print(f"Test acc: {model.evaluate(binary_2gram_test_ds)[1]:.3f}")
```

Модель достигла точности 90,4 % на контрольных данных — заметное улучшение! Оказывается, локальный порядок слов очень важен.

## Биграммы с кодированием TF-IDF

В представление можно добавить еще немного информации, подсчитав, сколько раз в тексте встречается каждое слово или  $N$ -грамма (иными словами, построив гистограмму распределения слов):

```
{"the": 2, "the cat": 1, "cat": 1, "cat sat": 1, "sat": 1,
 "sat on": 1, "on": 1, "on the": 1, "the mat": 1, "mat": 1}
```

Выполняя классификацию текста, важно знать, сколько раз слово встречается в выборке: любой достаточно длинный обзор фильма (независимо от общей оценки) может содержать слово «ужасный», но обзор, в котором слово «ужасный» употреблено много раз, скорее всего, отрицательный.

Вот как можно подсчитать количество вхождений биграмм с помощью слоя `TextVectorization`.

**Листинг 11.9.** Настройка слоя `TextVectorization` для подсчета токенов

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="count"
)
```

Очевидно, что некоторые слова будут встречаться в тексте чаще других независимо от его тематики. Слова `the`, `a`, `is` и `are` всегда будут доминировать в гистограммах, заглушая другие слова, несмотря на то что в контексте классификации они практически бесполезны. Можно ли решить данную проблему?

Вы уже наверняка предположили: да, можно — через нормализацию. Мы просто нормализуем количество слов, вычтя среднее значение и разделив разность на дисперсию (вычисленную по всему набору обучающих данных). В этом есть смысл. Вот только большинство векторизованных предложений почти полностью состоит из нулей (даже в нашем предыдущем примере имелось 12 ненулевых элементов и 19 988 нулевых). Подобное свойство называется разреженностью. Оно замечательное, поскольку значительно снижает вычислительную нагрузку и риск переобучения. А вычтя среднее из каждого признака, мы бы его разрушили. Поэтому в любой схеме нормализации мы можем применить только деление. Но что использовать в знаменателе? Лучшей практикой является то, что называется *нормализацией TF-IDF* (*term frequency, inverse document frequency* — «частота слова, обратная частота документа»).

Нормализация TF-IDF настолько распространена, что была встроена в слой `TextVectorization`. Чтобы задействовать ее, нужно просто передать в аргументе `output_mode` значение `"tf_idf"`.

### НОРМАЛИЗАЦИЯ TF-IDF

Чем чаще слово встречается в документе, тем важнее оно для понимания сути документа. В то же время частота употребления слова во всех документах в наборе данных тоже имеет значение. Если оно есть почти в каждом документе (как, например, *the* или *a*), оно малоинформативно. Но когда слово (скажем, *Herzog*) встречается в небольшом подмножестве текстов — это значит, что оно очень характерно и поэтому существенно для нас. Метрика TF-IDF объединяет эти две идеи. Она взвешивает заданное слово, беря частоту слова (то есть количество раз, которое оно встречается в текущем документе), и делит его на показатель «частота документа», оценивающий, как часто слово встречается во всем наборе данных. Вычисляется эта метрика следующим образом:

```
def tfidf(term, document, dataset):
    term_freq = document.count(term)
    doc_freq = math.log(sum(doc.count(term) for doc in dataset) + 1)
    return term_freq / doc_freq
```

**Листинг 11.10.** Настройка слоя `TextVectorization` на получение вывода, взвешенного метрикой TF-IDF

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="tf_idf",
)
```

Давайте обучим новую модель, используя эту схему.

**Листинг 11.11.** Обучение модели на биграммах, взвешенных метрикой TF-IDF, и ее проверка

```
text_vectorization.adapt(text_only_train_ds) ←
    Вызов adapt() обеспечит учет веса
    TF-IDF в дополнение к словарю
    tfidf_2gram_train_ds = train_ds.map(
        lambda x, y: (text_vectorization(x), y),
        num_parallel_calls=4)
    tfidf_2gram_val_ds = val_ds.map(
        lambda x, y: (text_vectorization(x), y),
        num_parallel_calls=4)
    tfidf_2gram_test_ds = test_ds.map(
        lambda x, y: (text_vectorization(x), y),
        num_parallel_calls=4)
```

```

model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("tfidf_2gram.keras",
                                    save_best_only=True)
]
model.fit(tfidf_2gram_train_ds.cache(),
          validation_data=tfidf_2gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("tfidf_2gram.keras")
print(f"Test acc: {model.evaluate(tfidf_2gram_test_ds)[1]:.3f}")

```

Модель достигла точности 89,8 % в задаче классификации IMDB — в данном случае взвешивание метрикой TF-IDF выглядит бесполезным. Однако во многих задачах классификации текстов при использовании TF-IDF часто можно отметить увеличение точности на один процентный пункт по сравнению с обычным бинарным кодированием.

### ЭКСПОРТ МОДЕЛИ, ОБРАБАТЫВАЮЩЕЙ ИСХОДНЫЕ СТРОКИ

В предыдущих примерах мы включили стандартизацию, разбиение и индексацию текста в конвейер `tf.data`. Но если понадобится экспортировать автономную модель, не зависящую от этого конвейера, то в нее следует включить свой механизм предварительной обработки текста (иначе вам придется заново реализовать этот механизм в промышленном окружении, что может добавить сложности или привести к мелким расхождениям между обучающими и производственными данными). К счастью, это легко сделать.

Просто создайте новую модель, повторно использующую слой `TextVectorization`, и добавьте в нее только что обученную модель:

```

inputs = keras.Input(shape=(1,), dtype="string") ←
processed_inputs = text_vectorization(inputs) ←
outputs = model(processed_inputs) ←
inference_model = keras.Model(inputs, outputs) ←
                                         ↓
                                         Создать экземпляр модели
  
```

Один входной образец  
должен быть строкой

Выполнить  
предварительную  
обработку текста

Добавить прежде  
обученную модель

Получившаяся модель способна обрабатывать пакеты исходных строк.

```

import tensorflow as tf
raw_text_data = tf.convert_to_tensor([
    ["That was an excellent movie, I loved it."],
])
predictions = inference_model(raw_text_data)
print(f"float(predictions[0] * 100):.2f} percent positive")

```

### 11.3.3. Обработка слов как последовательностей: модели последовательностей

Последние примеры наглядно демонстрируют, что порядок слов имеет значение: создание таких основанных на порядке признаков, как биграммы, дало хороший прирост точности. Теперь давайте вспомним, чему учит история глубокого обучения: необходимо отходить от ручного проектирования признаков и переходить к моделям, способным самостоятельно изучать признаки, основываясь только на данных. А что, если вместо конструирования вручную признаков, учитываяших порядок слов, передать модели необработанную последовательность слов и позволить ей самой определить наиболее информативные признаки? Именно так действуют *модели последовательностей*.

Чтобы реализовать модель последовательности, необходимо сначала представить входные образцы в виде последовательностей целочисленных индексов (одно целое число соответствует одному слову). Затем каждое целое число нужно отобразить в вектор, чтобы получить последовательности векторов. И наконец, эти последовательности векторов следует передать в стек слоев, которые смогут оценить коррелирующие признаки из соседних векторов, как это делают, например, одномерная сверточная сеть, рекуррентная нейронная сеть или архитектура Transformer.

Некоторое время тому назад (примерно в 2016–2017 годах) двунаправленные рекуррентные сети (в частности, двунаправленные сети LSTM) считались высшим достижением методики моделирования последовательностей. Вы уже знакомы с этой архитектурой — поэтому в наших первых примерах модели последовательности мы используем именно ее. Однако в настоящее время моделирование последовательностей почти повсеместно выполняется с помощью архитектуры Transformer, о которой мы поговорим ниже. Как ни странно, одномерные сверточные сети никогда не пользовались особой популярностью в обработке естественного языка, хотя, по моему собственному опыту, стек одномерных сверток с разделением по глубине и остаточными связями может достигать уровня, сопоставимого с двунаправленной сетью LSTM, при значительном снижении вычислительных затрат.

#### Первый практический пример

Давайте попробуем создать действующую модель последовательности. Для начала подготовим наборы данных, возвращающие целочисленные последовательности.

Затем сконструируем модель. Самый простой способ преобразовать целочисленные последовательности в векторные — прямое кодирование (каждое измерение будет представлять одно возможное слово в словаре). Поверх этих векторов прямого кодирования добавим простой двунаправленный слой LSTM.

**Листинг 11.12.** Подготовка наборов данных с целочисленными последовательностями

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y)),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

Чтобы сохранить контроль над размером входа, мы ограничиваем длину последовательностей 600 словами. Это разумный выбор, так как средняя длина отзыва составляет 233 слова, и только 5 % отзывов длиннее 600 слов

**Листинг 11.13.** Модель последовательности, принимающая последовательность векторов прямого кодирования

```
import tensorflow as tf
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Один вход — последовательность целых чисел

Кодируем целые числа в 20 000-мерные бинарные векторы

Добавляем двунаправленный слой LSTM

Наконец, добавляем слой классификации

Теперь обучим нашу модель.

**Листинг 11.14.** Обучение первой простой модели последовательности

```
callbacks = [
    keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("one_hot_bidir_lstm.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Первое наблюдение: эта модель обучается очень медленно, особенно в сравнении с облегченной моделью из предыдущего раздела. Причина в большом объеме входных данных: каждый входной образец кодируется как матрица с формой  $(600, 20000)$  (600 слов, 20 000 возможных слов) — а это 12 000 000 вещественных чисел на один обзор фильма. У нашей двунаправленной сети LSTM много работы. Второе наблюдение: модель достигла точности всего 87 % — значительно ниже нашей (очень быстрой) модели, обученной на бинарных унigramмах.

Очевидно, что простое использование прямого кодирования для преобразования слов в векторы оказалось не лучшей идеей. Но есть более удачный способ: векторные представления слов.

### **Векторные представления слов**

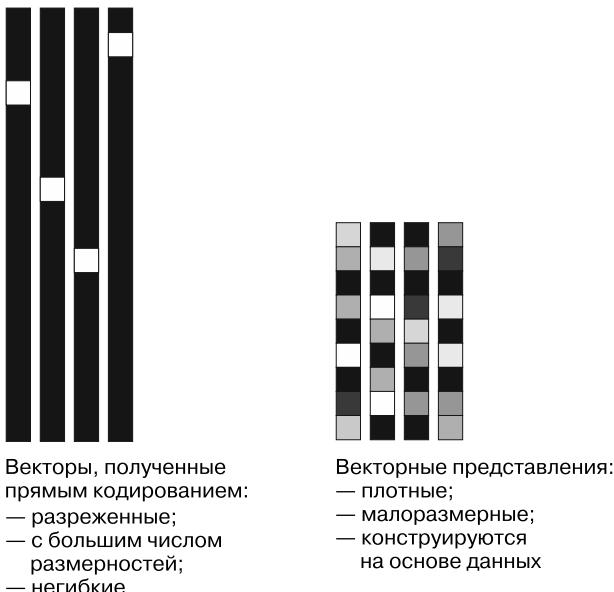
Важно отметить, что, выбирая метод прямого кодирования, вы принимаете решение из области конструирования признаков. Вы вводите в свою модель фундаментальное предположение о структуре пространства признаков, утверждая, что *разные токены не зависят друг от друга*, — и действительно, все векторы, получаемые методом прямого кодирования, друг другу ортогональны. Но в случае со словами это предположение явно неверное. Слова образуют структурированное пространство: они взаимно обмениваются информацией. Слова «кино» и «фильм» взаимозаменяемы в большинстве предложений, поэтому вектор, представляющий «кино», не ортогонален вектору, представляющему «фильм», — они должны быть одинаковыми или достаточно близкими.

Говоря более абстрактно, *геометрические отношения* между векторами должны отражать *семантические связи* между соответствующими им словами. Например, от правильно сконструированного пространства векторных представлений разумно ожидать, что синонимы будут представлены похожими векторами; и в целом геометрическое расстояние (такое как косинусное расстояние или L2-расстояние) между любыми двумя векторами будет зависеть от «семантического расстояния» между соответствующими словами. Слова с далеким друг от друга смыслом должны быть представлены далекими друг от друга точками, а слова со схожим смыслом — близкими.

*Векторные представления* (*word embeddings*) обладают этим свойством в полной мере: они отображают человеческий язык в геометрическое пространство.

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. 11.2. При работе с огромными словарями размерность векторов слов нередко может достигать 256, 512 или 1024. С другой стороны, прямое

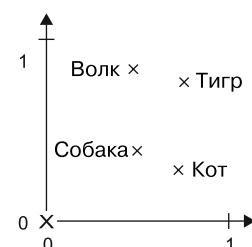
кодирование слов обычно влечет создание векторов с числом измерений 20 000 или больше (при использовании словаря с 20 000 токенов, как в данном случае). Иначе говоря, векторное представление слов позволяет уместить больший объем информации в меньшее число измерений.



**Рис. 11.2.** 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

Векторные представления слов не только *плотные*, но также структурированные, и их структура формируется на основе данных. Похожие слова не только оказываются поблизости друг от друга, но также наделяют определенным смыслом конкретные *направления* в пространстве векторов. Поясню это на примере.

На рис. 11.3 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



**Рис. 11.3.** Упрощенный пример векторного пространства слов

«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.

Давайте посмотрим, как использовать такое векторное пространство на практике. 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**

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 анализа 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 11.15. Создание слоя Embedding

```
embedding_layer = layers.Embedding(input_dim=max_tokens, output_dim=256) ←
    Слой Embedding принимает как минимум два аргумента: количество
    возможных токенов и размерность пространства (в данном случае 256)
```

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. 11.4).

Индекс слова → Слой Embedding → Вектор, соответствующий слову

**Рис. 11.4.** Слой Embedding

Слой **Embedding** получает на входе двумерный тензор с целыми числами и формой (**образцы, длина\_последовательности**), каждый элемент которого является последовательностью целых чисел, и возвращает трехмерный тензор с вещественными числами и формой (**образцы, длина\_последовательности, размерность\_векторного\_представления**).

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.

Давайте построим модель, включающую слой **Embedding**, и применим ее для решения нашей задачи.

#### Листинг 11.16. Модель, использующая слой Embedding, и ее обучение с нуля

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
```

```
metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Эта модель обучается намного быстрее модели с прямым кодированием (потому что слой LSTM обрабатывает лишь 256-мерные векторы вместо 20 000-мерных) и достигла сопоставимой точности на контрольных данных (87 %). Однако мы все еще далеки от уровня базовой биграммной модели. Причина отчасти заключается в том, что эта модель получает немного меньше данных: модель на основе биграмм обрабатывала полные обзоры, тогда как модель последовательностей ограничивала последовательности 600 словами.

### Заполнение и маскировка

Есть еще одно обстоятельство, вредящее модели, — наши входные последовательности полны нулей. Это обусловлено использованием параметра `output_sequence_length=max_length` в конструкторе слоя `TextVectorization` (со значением `max_length`, равным 600): отзывы длиннее 600 токенов усекаются до этой величины, а отзывы короче 600 токенов дополняются в конце нулями, чтобы их можно было объединить с другими последовательностями в непрерывные пакеты.

Мы используем двунаправленную рекуррентную сеть. Два рекуррентных слоя работают параллельно: один обрабатывает токены в их естественном порядке, а другой — те же токены в обратном порядке. Слой, работающий с токенами в естественном порядке, в своих последних итерациях будет обрабатывать векторы, почти полностью состоящие из заполняющих нулей, — таких итераций может быть несколько сотен, если исходный отзыв очень короткий. И информация, хранящаяся внутри рекуррентной сети, будет постепенно исчезать под влиянием бессмысленных входных данных.

Нам нужен какой-то способ сообщить слою, что он должен пропустить подобные итерации. И такой способ есть: API *маскирования*.

Слой `Embedding` способен сгенерировать маску, соответствующую его входным данным. Она имеет вид тензора из единиц и нулей (или логических значений `True/False`) с формой (`размер_пакета, длина_последовательности`), элемент `mask[i, t]` которого указывает, следует ли пропустить временной шаг `t` в образце `i` (временной шаг пропускается, если элемент `mask[i, t]` равен `0` или `False`, и обрабатывается в противном случае).

По умолчанию эта возможность отключена — ее можно включить, передав параметр `mask_zero=True` слою `Embedding`. Получить маску можно с помощью метода `calculate_mask()`:

```
>>> embedding_layer = Embedding(input_dim=10, output_dim=256, mask_zero=True)
>>> some_input = [
... [4, 3, 2, 1, 0, 0, 0],
... [5, 4, 3, 2, 1, 0, 0],
... [2, 1, 0, 0, 0, 0, 0]]
>>> mask = embedding_layer.compute_mask(some_input)
<tf.Tensor: shape=(3, 7), dtype=bool, numpy=
array([[ True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True,  True, False, False],
       [ True,  True,  False, False, False, False, False]])>
```

На практике вам редко придется управлять масками вручную: Keras автоматически передает маску каждому слову, готовому ее обработать (в виде фрагмента метаданных, прикрепленного к последовательности, которую он представляет). Эта маска будет использоваться рекуррентными слоями для пропуска замаскированных шагов. Если модель возвращает всю последовательность, то маску также применит функция потерь для пропуска замаскированных шагов в выходной последовательности.

Попробуем переобучить нашу модель с включенной маскировкой.

#### **Листинг 11.17.** Использование слоя `Embedding` с включенной маскировкой

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(
    input_dim=max_tokens, output_dim=256, mask_zero=True)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru_with_masking.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru_with_masking.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Модель достигла точности 88 % на контрольных данных — небольшое, но заметное улучшение.

## Использование предварительно обученных векторных представлений слов

Иногда обучающих данных оказывается слишком мало, чтобы можно было обучить векторное представление слов для конкретной задачи. В таких случаях 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 применение 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 — обобщенных визуальных и семантических. 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, первоначально была исследована Йошуа Бенгио с коллегами в начале 2000-х годов<sup>1</sup>, но более основательное ее изучение и практическое применение начались только после выхода одной из самых известных и успешных схем реализации векторного представления слов: алгоритма Word2Vec (<https://code.google.com/archive/p/word2vec>), разработанного в 2013 году Томасом Миколовым 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 году. 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. Начнем с загрузки файлов GloVe

---

<sup>1</sup> Bengio Y. et al. A Neural Probabilistic Language Model // Journal of Machine Learning Research, 2003.

и их анализа. Затем загрузим векторы слов в слой `Embedding`, который будем использовать для построения новой модели.

Прежде всего загрузим 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 (а также токенов, не являющихся словами).

```
!wget http://nlp.stanford.edu/data/glove.6B.zip  
!unzip -q glove.6B.zip
```

Обработаем распакованный файл `.txt` и создадим индекс, отображающий слова (в виде строк) в их векторные представления.

**Листинг 11.18.** Обработка файла с векторными представлениями слов GloVe

```
import numpy as np
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print(f"Found {len(embeddings_index)} word vectors.")
```

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 указателе (созданном в ходе токенизации).

**Листинг 11.19.** Подготовка матрицы векторных представлений слов GloVe

```
embedding_dim = 100          Получаем словарь, проиндексированный  
                             нашим предыдущим слоем TextVectorization  
  
vocabulary = text_vectorization.get_vocabulary()           ←  
word_index = dict(zip(vocabulary, range(len(vocabulary)))) ←  
  
embedding_matrix = np.zeros((max_tokens, embedding_dim))  
for word, i in word_index.items():  
    if i < max_tokens:  
        embedding_vector = embeddings_index.get(word)  
    if embedding_vector is not None:  
        embedding_matrix[i] = embedding_vector  
  
Используем его, чтобы  
создать словарь,  
отображающий  
слова в их индексы  
в vocabulary  
  
Подготавливаем  
матрицу  
для заполнения  
векторами GloVe  
  
Заполняем i-й элемент матрицы  
вектором слова с индексом i.  
Векторы для слов, не найденных  
в наборе векторных представлений,  
будут заполнены нулями
```

Наконец, мы используем инициализатор `Constant` для загрузки предварительно обученных векторных представлений в слой `Embedding`. Чтобы не разрушить эти представления во время обучения, заморозим слой, добавив параметр `trainable=False`:

```
embedding_layer = layers.Embedding(  
    max_tokens,  
    embedding_dim,  
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),  
    trainable=False,  
    mask_zero=True,  
)
```

Теперь мы готовы обучить новую модель — идентичную предыдущей, но использующую 100-мерные предварительно обученные векторные представления GloVe вместо 128-мерных представлений, сконструированных нами.

**Листинг 11.20.** Модель, использующая предварительно обученный слой Embedding

```
inputs = keras.Input(shape=(None,), dtype="int64")  
embedded = embedding_layer(inputs)  
x = layers.Bidirectional(layers.LSTM(32))(embedded)  
x = layers.Dropout(0.5)(x)  
outputs = layers.Dense(1, activation="sigmoid")(x)  
model = keras.Model(inputs, outputs)  
model.compile(optimizer="rmsprop",  
              loss="binary_crossentropy",  
              metrics=["accuracy"])  
model.summary()  
callbacks = [  
    keras.callbacks.ModelCheckpoint("glove_embeddings_sequence_model.keras",  
                                    save_best_only=True)  
]  
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,  
          callbacks=callbacks)  
model = keras.models.load_model("glove_embeddings_sequence_model.keras")  
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Увидев результаты, вы обнаружите, что в этой конкретной задаче предварительно обученные векторные представления оказались не особенно полезными: имеющийся в нашем распоряжении набор данных содержит достаточно образцов, чтобы построить в нужной степени специализированное пространство представлений с нуля. Однако использование предварительно обученных представлений может очень пригодиться, когда доступен лишь небольшой набор данных.

## 11.4. АРХИТЕКТУРА TRANSFORMER

Начиная с 2017 года вновь появившаяся архитектура моделей Transformer начала обгонять рекуррентные нейронные сети в большинстве задач обработки естественного языка.

Эта архитектура была представлена в основополагающей статье Васвани и его коллег *Attention is all you need*<sup>1</sup>. Суть статьи ясно выражена в ее названии<sup>2</sup>: как оказалось, с помощью простого механизма под названием «нейронное внимание» можно создавать мощные модели последовательностей, не имеющие ни повторяющихся, ни сверточных слоев.

Это открытие произвело настоящую революцию в обработке естественного языка и не только. Нейронное внимание быстро стало одной из доминирующих идей в глубоком обучении. Далее я подробно расскажу, как работает этот механизм и почему он оказался настолько эффективным при обработке последовательностей. Затем мы используем механизм внутреннего внимания (*self-attention*) для создания кодировщика Transformer — одного из основных компонентов архитектуры Transformer — и применим его к задаче классификации обзоров фильмов на IMDB.

### 11.4.1. Идея внутреннего внимания

Держа в руках эту книгу, вы можете бегло просматривать одни части и внимательно читать другие в зависимости от целей или интересов. А что, если ваши модели будут поступать так же? Это простая, но мощная идея: не вся входная информация одинаково важна для поставленной задачи, поэтому модели должны «уделять больше внимания» одним признакам и «меньше внимания» — другим.

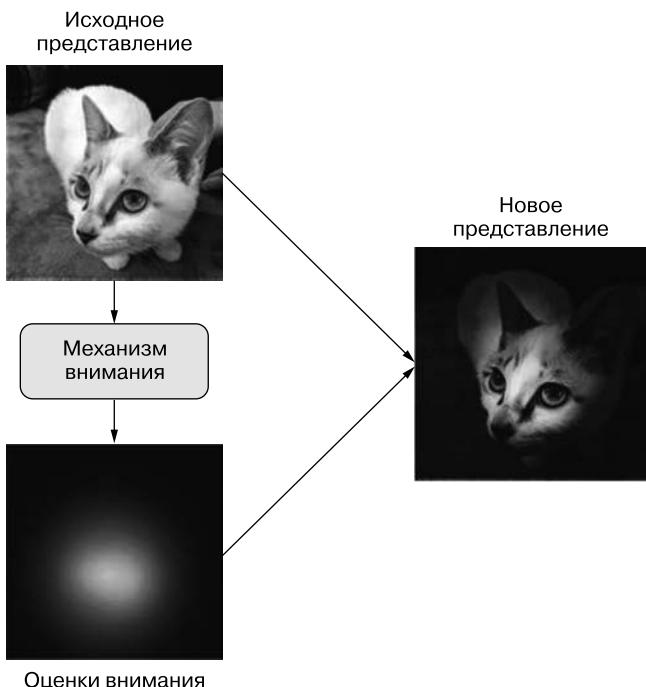
Звучит знакомо? Вы уже дважды встречались с подобной концепцией в этой книге:

- алгоритм выбора максимального из соседних значений в сверточных сетях просматривает пул значений в пространственной области и выбирает только одно из них. Эта форма внимания организована по принципу «все или ничего»: сохраняем самое важное значение и отказываемся от остальных;
- нормализация TF-IDF присваивает оценки важности токенам в зависимости от их информативности. Информативные токены усиливаются, а малоинформационные ослабляются. Это непрерывная форма внимания.

<sup>1</sup> Vaswani A. et al. Attention is all you need. 2017, <https://arxiv.org/abs/1706.03762>.

<sup>2</sup> «Внимание — это все, что вам нужно». — Примеч. пер.

Можно представить множество разных форм внимания, но все они начинаются с вычисления оценки важности набора признаков, при этом более высокие оценки присваиваются более релевантным признакам, а более низкие — менее релевантным (рис. 11.5). Порядок вычисления и использования оценок зависит от подхода.

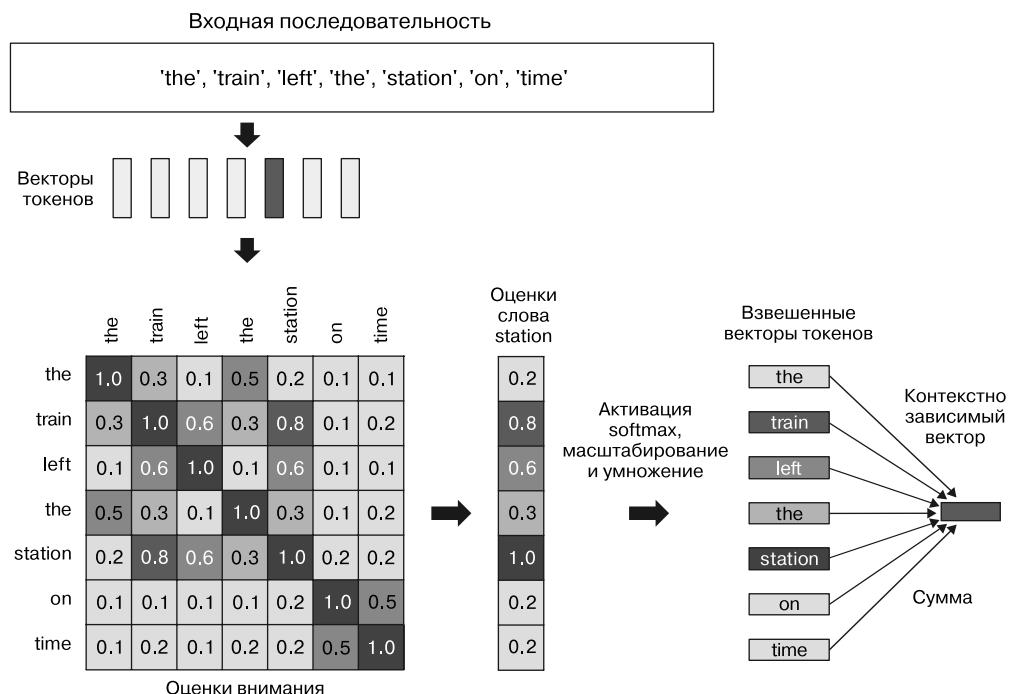


**Рис. 11.5.** Общая идея внимания в глубоком обучении:  
входным признакам присваиваются оценки внимания,  
которые можно использовать для уточнения следующего  
представления входных данных

Важно отметить, что такой механизм внимания можно использовать не только для усиления или ослабления определенных признаков, но также для создания контекстно зависимых признаков. Вы только что познакомились с векторными представлениями слов — векторными пространствами, фиксирующими «форму» семантических отношений между разными словами. В пространстве векторных представлений единственное слово имеет фиксированную позицию — фиксированный набор связей с каждым другим словом в пространстве. Но язык устроен не совсем так: значение слова обычно зависит от контекста. Описывая прогулку по речной косе, вы не имеете в виду «косу» — инструмент для скашивания травы

или женскую прическу. Когда вы говорите «скоро увидимся», то значение слова «увидимся» немного отличается от «видеть» во фразе «я вижу близкое окончание этого проекта» или «я вижу, куда вы клоните». И конечно же, значение таких местоимений, как «он», «оно», «она», и других полностью зависит от контекста и даже может меняться несколько раз в одном предложении.

Очевидно, что интеллектуальное пространство представлений придает разные векторные представления словам в зависимости от других окружающих его слов. Это и есть *внутреннее внимание (внимание к себе)*. Его цель — сформировать представление токена, используя представления связанных с ним токенов из последовательности. В результате получаются контекстно зависимые представления. Рассмотрим предложение *The train left the station on time* («Поезд ушел со станции вовремя»). Теперь рассмотрим одно слово из этого предложения: *station* («станция»). О какой станции идет речь? Может, это радиостанция? А может, Международная космическая станция? Разберемся с этим алгоритмически, использовав идею внутреннего внимания (рис. 11.6).



**Рис. 11.6.** Внутреннее внимание: оценки внимания вычисляются между словом *station* и каждым другим словом в последовательности и затем используются для взвешивания суммы векторов слов, которая становится новым вектором слова *station*

Шаг 1 — нужно вычислить оценки релевантности между вектором слова *station* и каждым другим словом в предложении. Это наши оценки внимания. В качестве меры силы связи слов мы будем использовать простое скалярное произведение их векторов. Это очень эффективная с вычислительной точки зрения функция расстояния, широко применявшаяся для оценки силы связи векторных представлений слов задолго до появления архитектуры Transformer. На практике данные оценки также могут подвергаться масштабированию и обработке функцией *softmax*, но все это лишь детали реализации.

Шаг 2 — вычисляем сумму всех векторов слов в предложении, взвешенных нашими оценками релевантности. Слова, тесно связанные со словом *station*, будут вносить больший вклад в сумму (включая само слово *station*), а нерелевантные слова почти ничего не дадут. Получившийся вектор — новое представление для *station*: представление, учитывающее окружающий контекст. В частности, оно включает часть вектора *train* (поезд), уточняя, что, по сути, речь идет о *train station* («вокзал»).

Тот же процесс можно повторить для каждого слова в предложении и создать новую последовательность векторов, кодирующих это предложение. Ниже приводится пример на псевдокоде NumPy реализации только что описанного алгоритма:

```
Выполнить итерации по всем токенам
во входной последовательности
def self_attention(input_sequence):
    output = np.zeros(shape=input_sequence.shape)
    for i, pivot_vector in enumerate(input_sequence):
        scores = np.zeros(shape=(len(input_sequence),))
        for j, vector in enumerate(input_sequence):
            scores[j] = np.dot(pivot_vector, vector.T)
        scores /= np.sqrt(input_sequence.shape[1])
        scores = softmax(scores)
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)
        for j, vector in enumerate(input_sequence):
            new_pivot_representation += vector * scores[j]
        output[i] = new_pivot_representation
    return output
Вычислить скалярное
произведение (оценку
внимания) между
данным токеном
и каждым другим
токеном
Получить сумму векторов
всех токенов, взвешенных
оценками внимания
Масштабировать с использованием коэффициента
нормализации и применением функции softmax
Эта сумма и является
результатом
```

Конечно, на практике вы бы применили векторную реализацию. В Keras для этого имеется встроенный слой: *MultiHeadAttention*. Вот как можно его использовать:

```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

У многих из вас наверняка возникли вопросы.

- Почему входные данные передаются слою в *трех* параметрах? Это выглядит излишним.
- О какой многоголовости (multi-head) идет речь? Звучит пугающе — они что, отрастают, если их обрезать?

На оба вопроса есть простые ответы. Давайте посмотрим.

### **Обобщенное внутреннее внимание: модель «запрос — ключ — значение»**

До сих пор мы рассматривали только одну входную последовательность. Однако архитектура Transformer изначально разрабатывалась для машинного перевода, где приходится иметь дело с двумя входными последовательностями: исходной, которая переводится (например, «Какая сегодня погода?»), и целевой, в которую должна быть преобразована текущая (например, *¿Qué tiempo hace hoy?*). Архитектура Transformer — это модель типа «последовательность в последовательность»: она была разработана для преобразования одной последовательности в другую. Далее в этой главе вы познакомитесь ближе с моделями последовательностей.

Но пока давайте сделаем шаг назад. Механизм внутреннего внимания, как мы его представили, в общих чертах выполняет следующее:

```
outputs = sum(inputs * pairwise_scores (inputs, inputs))
```



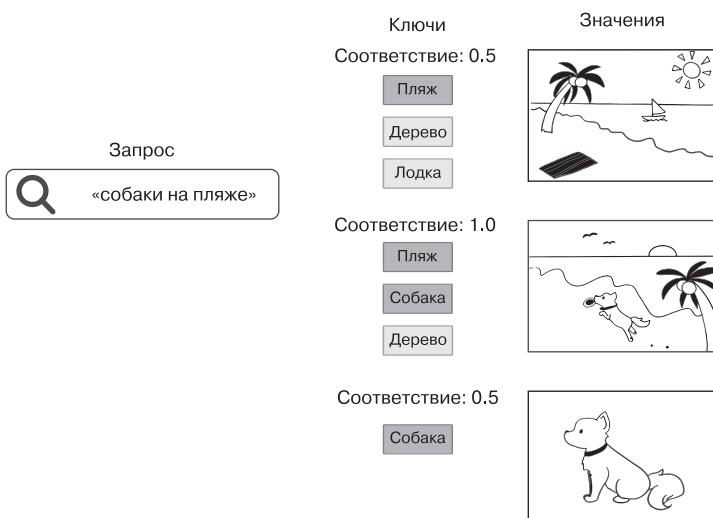
Это означает: «для каждого токена в `inputs` (A) определить, насколько сильно он связан с каждым токеном в `inputs` (B), и использовать полученные оценки для взвешивания суммы токенов из `inputs` (C)». Важно отметить, что A, B и C не обязательно должны ссылаться на одну и ту же входную последовательность. В общем случае это могут быть три разных последовательности. Назовем их «запрос» (query), «ключи» (keys) и «значения» (values). В таком случае смысл операции выражается следующим образом: «для каждого токена в запросе определить, насколько сильно он связан с каждым ключом, и использовать полученные оценки для взвешивания суммы значений»:

```
outputs = sum(values * pairwise_scores(query, keys))
```

Данная терминология пришла из поисковых и рекомендательных систем (рис. 11.7). Представьте, что вы вводите запрос «собаки на пляже» для выбора фотографий из вашей коллекции. Каждая фотография в вашей базе данных описывается набором ключевых слов — «кошка», «собака», «вечеринка» и т. д.

Будем называть их ключами. Поисковая система сначала сравнивает запрос с ключами в базе данных. Совпадение со словом «собака» дает оценку соответствия 1, а отсутствие совпадения со словом «кошка» дает оценку соответствия 0. Затем она ранжирует ключи по величине соответствия (релевантности) и возвращает фотографии, связанные с лучшими  $N$  совпадениями в порядке убывания релевантности.

Концептуально именно так работает механизм внимания в архитектуре Transformer. У вас есть исходная последовательность, характеризующая искомое, — запрос. Есть совокупность знаний, из которых извлекается информация — значения. Каждому значению соответствует ключ, описывающий значение в формате, пригодном для сравнения с запросом. Вы просто сопоставляете запрос с ключами и возвращаете взвешенную сумму значений.



**Рис. 11.7.** Извлечение изображений из базы данных: запрос сравнивается с набором ключей, а оценки соответствия используются для ранжирования значений (изображений)

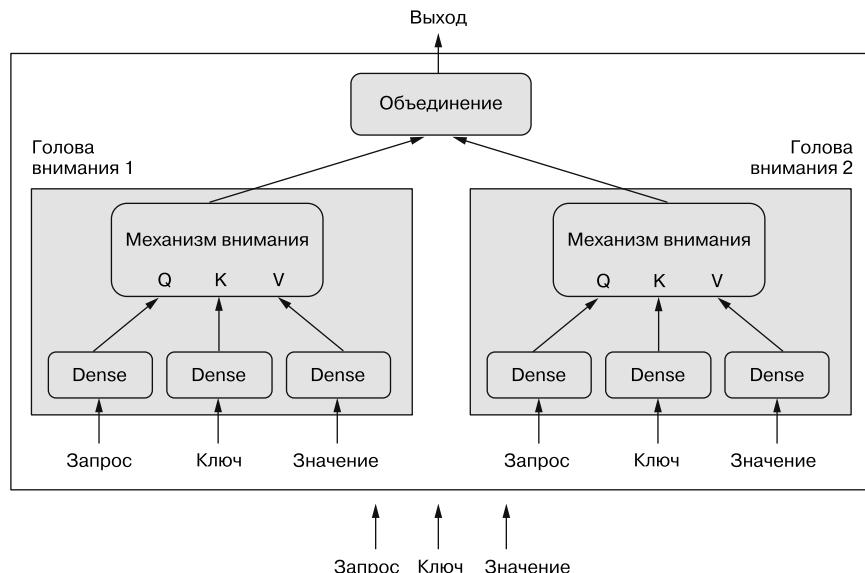
На практике ключи и значения часто являются одинаковыми последовательностями. Например, в машинном переводе запрос может быть целевой последовательностью, а исходная последовательность — как ключами, так и значениями: для каждого элемента в целевой последовательности (например, *tiempo*) нужно вернуться к исходной последовательности («Какая сегодня погода?») и определить различные элементы, связанные с ним (*tiempo* и «погода» должны иметь точное соответствие). Естественно, в простой задаче классификации последовательности запрос, ключи и значения — это одно и то же: последовательность

сравнивается с самой собой, чтобы обогатить каждый токен контекстом из всей последовательности.

Это объясняет, почему потребовалось передать `inputs` слою `MultiHeadAttention` в трех параметрах. Но почему внимание многоголовое (multi-head)?

### 11.4.2. Многоголовое внимание

Многоголовое внимание (multi-head attention) — это дополнительная настройка механизма внутреннего внимания, представленного в статье *Attention is all you need*. Под многоголовостью подразумевается разделение выходного пространства слоя внутреннего внимания на набор независимых подпространств, обучаемых отдельно: исходный запрос, ключ и значение передаются через три независимых набора плотных проекций, в результате чего получается три отдельных вектора. Каждый вектор обрабатывается с привлечением механизма нейронного внимания, после чего три выхода снова объединяются в единую выходную последовательность. Каждое такое подпространство называется головой. Полная картина процесса показана на рис. 11.8.



**Рис. 11.8.** Слой MultiHeadAttention

Наличие обучаемых плотных проекций позволяет слою действительно чему-то учиться — в отличие от преобразования без сохранения состояния, для которого

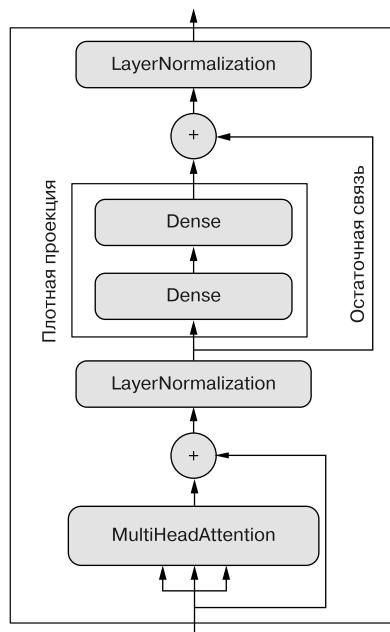
требуются дополнительные слои, расположенные до или после него, чтобы преобразование приносило пользу. Кроме того, наличие независимых голов помогает слову изучать разные группы признаков для каждого токена, где признаки в одной группе коррелируют друг с другом, но практически не зависят от признаков в другой группе.

В принципе, это похоже на то, как работают свертки с разделением по глубине: в такой свертке выходное пространство разбивается на множество подпространств (по одному для каждого входного канала), изучаемых независимо. Статья *Attention is all you need* была написана в ту пору, когда на практике было показано, что идея деления пространств признаков на независимые подпространства дает существенные преимущества моделям компьютерного зрения — как в случае сверток с разделением по глубине, так и в случае близких к ним *сгруппированных сверток*. Многоголовое внимание — это просто применение той же идеи к внутреннему вниманию.

### 11.4.3. Кодировщик Transformer

Если добавление дополнительных плотных проекций так полезно, то почему бы не применить еще пару к выходу механизма внимания? На самом деле это отличная идея — давайте так и поступим. Теперь модель стала довольно глубокой, поэтому можно добавить остаточные связи, чтобы не уничтожить ценную информацию в процессе обработки, — как вы узнали в главе 9, остаточные связи необходимы в любой глубокой архитектуре. Кроме этого, из главы 9 вы должны помнить следующее: предполагается, что слои нормализации способствуют передаче градиентов во время обратного распространения. Добавим и их.

Примерно такой мыслительный процесс, как мне представляется, развернулся в свое время в умах изобретателей архитектуры Transformer. Разделение выходных данных по нескольким независимым пространствам, добавление остаточных связей и слоев нормализации — все это стандартные архитектурные шаблоны, которые разумно использовать в любой сложной модели. Все вместе они образуют *кодировщик Transformer* — одну из двух важнейших частей архитектуры Transformer (рис. 11.9).



**Рис. 11.9.** TransformerEncoder объединяет слой MultiHeadAttention с плотной проекцией, а также добавляет нормализацию и остаточные связи

Оригинальная архитектура Transformer состоит из двух частей: *кодировщика Transformer*, обрабатывающего входную последовательность, и *декодера Transformer*, использующего входную последовательность для создания преобразованной версии. С устройством декодера вы познакомитесь чуть ниже.

Важно отметить, что кодировщик можно использовать для классификации текста — это универсальный модуль, который принимает последовательность и учится преобразовывать ее в более полезное представление. Давайте реализуем кодировщик Transformer и опробуем его на задаче классификации обзоров фильмов.

**Листинг 11.21.** Кодировщик Transformer реализован как подкласс класса Layer

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim ← Размер входного вектора токенов
        self.dense_dim = dense_dim ← Размер внутреннего полно связного слоя
        self.num_heads = num_heads ← Количество голов внимания
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),])
    self.layernorm_1 = layers.LayerNormalization()
    self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None): ← Вычисления выполняются в методе call()
        if mask is not None: ← Маска, слой Embedding сгенерирует
            mask = mask[:, tf.newaxis, :] двумерную маску, но слой внимания
        attention_output = self.attention( должен быть трех- или четырехмерным,
            inputs, inputs, attention_mask=mask) поэтому мы увеличиваем ранг
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self): ← Реализует сериализацию, чтобы дать
        config = super().get_config() возможность сохранить модель
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

## СОХРАНЕНИЕ НЕСТАНДАРТНЫХ СЛОЕВ

При создании своих слоев обязательно реализуйте метод `get_config`: это позволяет восстановить слой из его конфигурационного словаря, что пригодится при сохранении и загрузке модели. Метод должен возвращать словарь Python со значениями аргументов конструктора, использовавшихся для создания слоя.

Все слои Keras можно сериализовать и десериализовать, как показано ниже:

```
config = layer.get_config()
new_layer = layer.__class__.from_config(config) ← Конфигурация не содержит
                                                значений весов,
                                                поэтому все веса в слое
                                                инициализируются с нуля
```

Например:

```
layer = PositionalEmbedding(sequence_length, input_dim, output_dim)
config = layer.get_config()
new_layer = PositionalEmbedding.from_config(config)
```

При сохранении модели с нестандартными слоями файл будет содержать эти конфигурационные словари. При загрузке модели из файла вы должны передать нестандартные классы слоев методу загрузки, чтобы он мог правильно воссоздать объекты, упомянутые в конфигурации:

```
model = keras.models.load_model(
    filename, custom_objects={"PositionalEmbedding": PositionalEmbedding})
```

Обратите внимание, что здесь для нормализации применяются слои, отличные от `BatchNormalization`, которые мы использовали в моделях обработки изображений. Причина в том, что `BatchNormalization` плохо работает с последовательностями. Поэтому в данном случае берется слой `LayerNormalization`, который нормализует каждую последовательность независимо от других последовательностей в пакете, как показано в NumPy-подобном псевдокоде:

```
Входные данные имеют форму
(размер_пакета, длина_последовательности,
размерность_векторного_представления) ←

def layer_normalization(batch_of_sequences): ← Для вычисления среднего
    mean = np.mean(batch_of_sequences, keepdims=True, axis=-1)   и дисперсии мы объединяем данные
    variance = np.var(batch_of_sequences, keepdims=True, axis=-1) только по последней оси (-1)
    return (batch_of_sequences - mean) / variance
```

Сравните с тем, как действует слой `BatchNormalization` (на этапе обучения):

```
def batch_normalization(batch_of_images): ←
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
    return (batch_of_images - mean) / variance
```

В отличие от слоя `BatchNormalization`, который собирает информацию по множеству образцов, чтобы получить точные значения среднего и дисперсии признаков, слой `LayerNormalization` объединяет данные по последовательностям и лучше подходит для их обработки.

Теперь, реализовав `TransformerEncoder`, используем его для сборки модели классификации текста, похожей на модель на основе GRU, которую вы видели ранее.

#### Листинг 11.22. Использование кодировщика Transformer для классификации текста

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x) ←
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Обучим получившуюся модель. Она достигла точности 87,5 % — чуть хуже модели GRU.

#### Листинг 11.23. Обучение и оценка модели на основе кодировщика Transformer

```
callbacks = [
    keras.callbacks.ModelCheckpoint("transformer_encoder.keras",
        save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
          callbacks=callbacks)
model = keras.models.load_model(
    "transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder}) ←
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Сейчас у вас должно появиться чувство тревоги: что-то явно пошло не так. Сможете ли вы определить, что именно?

Данный раздел посвящен моделям последовательностей. В самом начале я подчеркнул важность порядка слов и заявил, что Transformer — это архитектура обработки последовательностей, изначально разработанная для машинного перевода. И вместе с тем... кодировщик Transformer, который вы только что видели в действии, вообще не является моделью последовательностей. Заметили это? Он состоит из полносвязных слоев, обрабатывающих последовательности токенов независимо друг от друга, и слоя внимания, который рассматривает токены *как множество*. Изменив порядок токенов в последовательности, вы получите точно такие же попарные оценки внимания и точно такие же контекстно зависимые представления. Даже если перемешать все слова в каждом обзоре фильма, модель этого не распознает и вы все равно получите ту же самую точность. Внутреннее внимание — это механизм обработки множеств, сосредоточенный на отношениях между парами элементов последовательности (рис. 11.10). Он никак не учитывает места, где встречаются эти элементы: в начале, в конце или в середине последовательности. Так почему же мы называем архитектуру Transformer моделью последовательности? И как она может использоваться для машинного перевода, если не учитывает порядок слов?

	Учитывает порядок слов	Учитывает контекст (взаимосвязи между словами)
Мешок униграмм	Нет	Нет
Мешок bigramm	Очень ограниченно	Нет
Рекуррентная нейронная сеть	Да	Нет
Механизм внутреннего внимания	Нет	Да
Transformer	Да	Да

**Рис. 11.10.** Свойства разных типов моделей обработки естественного языка

Выше я мимоходом подсказывал решение, упомянув, что Transformer — это гибридный подход, который технически не учитывает порядка элементов последовательностей, но позволяет вручную ввести информацию о порядке в обрабатываемые представления. Данный недостающий ингредиент называется *позиционным кодированием*. Давайте рассмотрим его.

## Использование позиционного кодирования для внедрения информации о порядке

Идея позиционного кодирования проста: чтобы модель могла получить информацию о порядке слов, мы добавим в каждое векторное представление каждого слова его позицию в предложении. Наши входные векторные представления слов

будут состоять из двух компонентов: обычного вектора, представляющего слово независимо от конкретного контекста, и вектора, представляющего положение слова в текущем предложении. Будем надеяться, что модель поймет, как лучше использовать это дополнение.

Самая простая схема, которую можно придумать, — соединить позицию слова с его векторным представлением, добавив в вектор ось «положения» и заполнив ее нулями для первого слова, единицами — для второго и т. д.

Однако это не наилучшее решение, поскольку позиции могут быть очень большими целыми числами, выходящими за границы диапазона значений в векторе представления. Как вы уже знаете, нейронные сети не любят огромных входных значений или дискретных распределений.

В статье *Attention is all you need* для кодирования позиций слов предлагался интересный прием: к векторным представлениям слов добавлялся вектор со значениями в диапазоне  $[-1, 1]$ , которые циклически менялись в зависимости от позиции (для этого использовалась функция косинуса). Данный трюк позволяет однозначно охарактеризовать любое целое число в большом диапазоне с помощью вектора малых значений. Любопытное решение, но в нашем случае мы к нему не прибегнем. Мы поступим проще и эффективнее: будем конструировать векторы представления позиций точно так же, как конструируем индексы векторных представлений слов. После этого мы добавим представления позиций к соответствующим представлениям слов, чтобы получить векторные представления слов с учетом их положения. Данный метод называется векторным представлением позиций. Давайте реализуем это решение.

#### Листинг 11.24. Реализация векторного представления позиций в подклассе слоя

```
Недостаток подхода с использованием векторных представлений позиций — в необходимости заранее знать длину последовательности
```

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim) ← Подготовим один слой Embedding для обработки индексов токенов
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim) ← А другой — для обработки позиций токенов
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions ← Объединим два вектора представлений
```

```

def compute_mask(self, inputs, mask=None):
    return tf.math.not_equal(inputs, 0)

→ def get_config(self):
    config = super().get_config()
    config.update({
        "output_dim": self.output_dim,
        "sequence_length": self.sequence_length,
        "input_dim": self.input_dim,
    })
    return config

```

Реализует сериализацию, чтобы дать возможность сохранить модель. Так же как слой Embedding, этот слой должен уметь генерировать маску, чтобы дать возможность игнорировать дополнение входных данных нулями. Фреймворк автоматически вызовет метод calculate\_mask, и полученная маска будет передана следующему слою

Слой PositionEmbedding можно использовать так же, как обычный слой Embedding. Посмотрим на него в действии!

## Классификация с использованием кодировщика Transformer

Чтобы начать учитывать порядок слов, нужно заменить старый слой Embedding нашей версией, отслеживающей их положение.

### Листинг 11.25. Объединение кодировщика Transformer с векторным представлением позиций

```

vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs) ← Смотрите сюда!
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
          callbacks=callbacks)
model = keras.models.load_model(
    "full_transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder,
                   "PositionalEmbedding": PositionalEmbedding})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")

```

Модель достигла точности 88,3 % на контрольных данных — серьезное улучшение, ясно демонстрирующее ценность информации о порядке слов для классификации текста. На данный момент это наша лучшая модель последовательностей! И все же она на ступень ниже модели мешка слов.

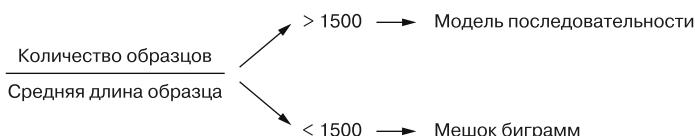
#### 11.4.4. Когда использовать модели последовательностей вместо моделей мешка слов

Иногда можно услышать, что методы мешка слов устарели и вместо них следует брать модели последовательностей на основе архитектуры Transformer, независимо от решаемой задачи или используемого набора данных. Это определенно неверно: небольшой стек слоев Dense и мешок биграмм все еще актуальны при решении многих задач. На самом деле среди разных методов, опробованных нами на наборе данных IMDB в этой главе, самым лучшим пока что был мешок биграмм!

Так когда лучше использовать тот или иной подход?

В 2017 году мы с коллегами проанализировали эффективность различных методов классификации текста на множестве наборов текстовых данных разных типов и вывели удивительное правило, помогающее решить, когда использовать модель мешка слов, а когда модель последовательности (<http://mng.bz/AOzK>) — своего рода золотую константу классификации.

Приступая к решению новой задачи классификации текста, следует обратить пристальное внимание на соотношение между количеством образцов в обучающих данных и средним количеством слов в образце (рис. 11.11). Если это соотношение невелико — менее 1500, лучше взять модель мешка биграмм (к тому же она будет намного быстрее обучаться и выполнять итерации). Если соотношение больше 1500, следует использовать модель последовательности. Иначе говоря, модели последовательностей работают лучше, когда доступно много обучающих данных, а образцы относительно короткие.



**Рис. 11.11.** Простая эвристика для выбора модели классификации текста: отношение количества обучающих образцов к среднему количеству слов в образце

Поэтому для задачи классификации, когда обучающая выборка включает 100 000 документов со средней длиной 1000 слов (отношение 100), лучше использовать модель биграмм. Она также будет полезна в задаче классификации твитов, в среднем состоящих из 40 слов, при обучающей выборке в 50 000 твитов (отношение 1250). Но если размер выборки увеличится до 500 000 твитов

(отношение 12 500), предпочтительнее взять кодировщик Transformer. Взглянем на задачу классификации отзывов к фильмам IMDB в свете этой закономерности. У нас имелось 20 000 обучающих образцов с 233 словами в среднем, поэтому согласно нашему эмпирическому правилу стоило использовать модель bigram — что и подтвердилось на практике.

Причина этого интуитивно понятна: входные данные моделей последовательностей определяют более богатое и сложное пространство, поэтому необходимо больше данных для его отображения; простое множество (мешок) терминов — это настолько простое пространство, что на нем можно обучить логистическую регрессию, используя всего несколько сотен или тысяч образцов. Кроме того, чем короче образцы, тем в меньшей степени модель может позволить себе отбрасывать какую-либо информацию — в частности, порядок слов приобретает дополнительную важность и его отбрасывание может создать двусмысленность. Предложения *This movie is the bomb* и *This movie was a bomb*<sup>1</sup> имеют очень близкие представления униграмм, что может запутать модель мешка слов, но модель последовательностей точно укажет, какое из них отрицательное, а какое положительное. С увеличением длины образцов статистика по словам становится более надежной, а тема или эмоциональная окраска — более очевидными.

Имейте в виду, что наше правило было разработано специально для классификации текстов. Оно может не соответствовать другим задачам обработки естественного языка — например, в машинном переводе архитектура Transformer справляется с очень длинными последовательностями намного лучше, чем рекуррентная сеть. Кроме того, данная эвристика является лишь эмпирическим правилом, а не научным законом, поэтому можно ожидать, что она будет работать в большинстве случаев, но не во всех.

## 11.5. ЗА ГРАНИЦАМИ КЛАССИФИКАЦИИ ТЕКСТА: ОБУЧЕНИЕ «ПОСЛЕДОВАТЕЛЬНОСТЬ В ПОСЛЕДОВАТЕЛЬНОСТЬ»

Теперь у вас есть все необходимое для решения большинства задач обработки естественного языка. Однако вы видели практическое применение этих инструментов только на примере одной задачи: классификации текста. Это чрезвычайно популярный случай; тем не менее сфера обработки естественного языка гораздо шире и включает не только классификацию. Далее вы приобретете дополнительный опыт, познакомившись с моделями типа «последовательность в последовательность».

Модель «последовательность в последовательность» принимает последовательность на входе (например, предложение или абзац) и преобразует ее в другую

---

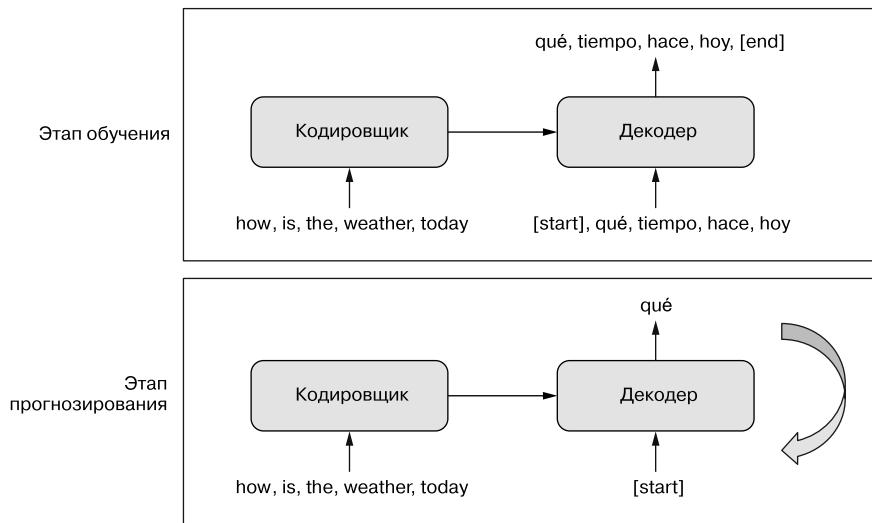
<sup>1</sup> Словосочетание *a bomb* носит отрицательный оттенок в отношении кино, *a the bomb*, на-против, выражает восхищение. Фраза *This movie was a bomb* переводится как «Этот фильм потерпел крах», а *This movie is the bomb* — как «Этот фильм — бомба!». — Примеч. пер.

последовательность. Данная задача лежит в основе многих успешных приложений обработки естественного языка:

- *машинный перевод* — преобразование абзаца на исходном языке в его эквивалент на целевом языке;
- *обобщение текста* — преобразование длинного документа в более короткую версию с сохранением наиболее важной информации;
- *ответы на вопросы* — преобразование входного вопроса в ответ на него;
- *чат-боты* — преобразование фразы собеседника в диалоге в ответ на нее или всего предыдущего диалога в следующую реплику;
- *генерация текста* — преобразование текстовой подсказки в абзац, завершающий эту подсказку, и др.

Общий шаблон моделей «последовательность в последовательность» показан на рис. 11.12. Во время обучения:

- модель *кодировщика* превращает исходную последовательность в промежуточное представление;
- *декодер* обучается предсказывать следующий токен *i* в целевой последовательности, просматривая предыдущие токены (от 0 до *i* - 1) и закодированную исходную последовательность.



**Рис. 11.12.** Обучение типа «последовательность в последовательность»: исходная последовательность обрабатывается кодировщиком и затем отправляется в декодер. Декодер просматривает целевую последовательность, созданную к настоящему моменту, и прогнозирует следующий шаг в будущем. Во время прогнозирования модель генерирует целевые токены по одному и возвращает их обратно в декодер

Во время прогнозирования у нас нет доступа к целевой последовательности — мы пытаемся предсказать ее с нуля, поэтому ее придется генерировать по одному токену за раз.

1. Сначала мы получаем от кодировщика закодированную исходную последовательность.
2. Декодер начинает с просмотра закодированной исходной последовательности, а также начального токена (например, строки "[start]"), и использует их для предсказания первого токена в выходной последовательности.
3. Прогнозируемая в данный момент последовательность возвращается обратно в декодер, который генерирует следующий токен, и т. д., пока не будет сгенерирован конечный токен (например, строка "[end]").

Для создания модели нового типа вам пригодится все, что вы узнали к настоящему моменту. Углубимся в детали.

### 11.5.1. Пример машинного перевода

Рассмотрим моделирование типа «последовательность в последовательность» на примере задачи машинного перевода. Именно для нее была разработана архитектура Transformer! Начнем с рекуррентной модели последовательности и затем перейдем к полной архитектуре Transformer.

За основу возьмем набор данных переводов с английского языка на испанский, доступный по адресу [www.manythings.org/anki/](http://www.manythings.org/anki/). Загрузим его:

```
!wget http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip
!unzip -q spa-eng.zip
```

Текстовый файл содержит по одному примеру в строке: предложение на английском языке, за которым следует символ табуляции и соответствующее предложение на испанском языке. Разобьем этот файл.

```
text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
    Обход всех строк в файле
for line in lines:
    english, spanish = line.split("\t")
        ← Каждая строка содержит фразу
        на английском языке, за которой следует
        символ табуляции и затем — эквивалентная
        фраза на испанском языке
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

← В начало и в конец фразы  
на испанском языке добавим токены  
"[start]" и "[end]" соответственно,  
как отмечено на рис. 11.12

Вот как должно выглядеть содержимое `text_pairs`:

```
>>> import random
>>> print(random.choice(text_pairs))
("Soccer is more popular than tennis.",
 "[start] El fútbol es más popular que el tenis. [end]")

```

Теперь перемешаем пары и разобьем их, как обычно, на обучающий, проверочный и контрольный наборы:

```
import random
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

Далее подготовим два отдельных слоя `TextVectorization`: один для английского и один для испанского языка. Нужно настроить предварительную обработку строк:

- мы должны сохранить добавленные нами токены "[start]" и "[end]"; по умолчанию символы [ и ] будут удаляться, но их следует оставить, чтобы различать слово *start* и начальный токен "[start]";
- в разных языках используются разные правила пунктуации! В слое `TextVectorization` для испанского языка, если будут удаляться знаки препинания, также следует предусмотреть удаление символа ꝑ.

Обратите внимание, что в реальной модели перевода знаки препинания следовало бы рассматривать как отдельные токены, а не удалять их — желательно, чтобы модель генерировала предложения с правильной пунктуацией. Но в нашем примере ради простоты мы избавимся от них.

#### **Листинг 11.26.** Векторизация пар предложений на английском и испанском языках

```
import tensorflow as tf
import string
import re

strip_chars = string.punctuation + " "
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")
```

Подготовка функции стандартизации строк для слоя `TextVectorization`, обрабатывающего текст на испанском языке: она должна сохранить квадратные скобки [ и ], но удалить ꝑ (а также все другие символы из `string.punctuation`)

```
source_vectorization = layers.TextVectorization(  
    max_tokens=vocab_size,  
    output_mode="int",  
    output_sequence_length=sequence_length,  
)  
target_vectorization = layers.TextVectorization(  
    max_tokens=vocab_size,  
    output_mode="int",  
    output_sequence_length=sequence_length + 1,  
    standardize=custom_standardization,  
)  
train_english_texts = [pair[0] for pair in train_pairs]  
train_spanish_texts = [pair[1] for pair in train_pairs]  
source_vectorization.adapt(train_english_texts)  
target_vectorization.adapt(train_spanish_texts)
```

Для простоты ограничим словарь 15 000 наиболее часто используемых слов в каждом языке, а длину предложений — 20 словами

Слой для обработки строк на английском языке

Слой для обработки строк на испанском языке

Предложения на испанском языке следует генерировать с одним дополнительным токеном, потому что во время обучения нужно сместить предложение на один шаг

Конструирование словаря для каждого языка

Наконец, мы преобразуем наши данные в конвейер `tf.data`, возвращающий кортеж (`inputs`, `target`), где `inputs` — это словарь с двумя ключами: вход для кодировщика (предложение на английском) и вход для декодера (предложение на испанском), а `target` — предложение на испанском, смещённое на один шаг вперед.

### Листинг 11.27. Подготовка наборов данных для задачи машинного перевода

```
batch_size = 64

def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({
        "english": eng,
        "spanish": spa[:, :-1],
    }, spa[:, 1:])

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset, num_parallel_calls=4)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

Входное предложение на испанском языке не включает последний токен, чтобы входные данные и цели имели одинаковую длину

Целевое испанское предложение на один шаг впереди. Оба имеют одинаковую длину (20 слов)

Применение кеширования в памяти для увеличения скорости обработки

Вот как выглядит наш набор данных:

```
>>> for inputs, targets in train_ds.take(1):
>>>     print(f"inputs['english'].shape: {inputs['english'].shape}")
>>>     print(f"inputs['spanish'].shape: {inputs['spanish'].shape}")
>>>     print(f"targets.shape: {targets.shape}")
inputs["encoder_inputs"].shape: (64, 20)
inputs["decoder_inputs"].shape: (64, 20)
targets.shape: (64, 20)
```

Данные готовы — можно начинать строить модели. Начнем с рекуррентной модели последовательностей, а затем перейдем к Transformer.

## 11.5.2. Обучение типа «последовательность в последовательность» рекуррентной сети

Рекуррентные нейронные сети доминировали в обучении типа «последовательность в последовательность» с 2015 по 2017 год, прежде чем их обогнала архитектура Transformer. Они служили основой многим системам машинного перевода — как упоминалось в главе 10, система перевода Google Translate примерно в 2017 году работала на стеке из семи больших слоев LSTM. Данный подход сохраняет свою актуальность и по сей день, поскольку это простая отправная точка в освоении моделей «последовательность в последовательность».

Самый простой способ использовать рекуррентную сеть для преобразования одной последовательности в другую — сохранять выходные данные сети на каждом временном шаге. В Keras это будет выглядеть так:

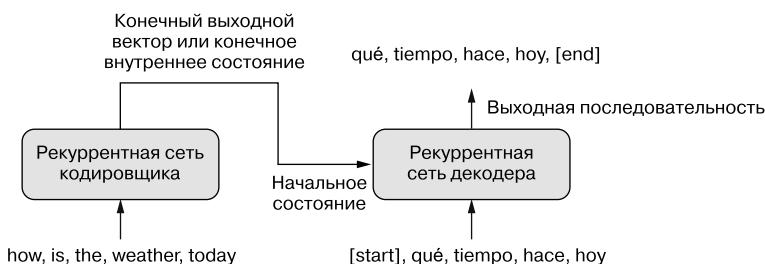
```
inputs = keras.Input(shape=(sequence_length,), dtype="int64")
x = layers.Embedding(input_dim=vocab_size, output_dim=128)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
```

Однако при таком подходе возникает две проблемы:

- целевая последовательность должна иметь ту же длину, что и исходная. Но в действительности так бывает очень редко. С технической точки зрения это не очень критично: всегда можно дополнить исходную или целевую последовательность так, чтобы их длины совпадали;
- из-за пошаговой природы рекуррентных сетей модель будет рассматривать только токены  $0 \dots N$  в исходной последовательности, чтобы предсказать токен  $N$  в целевой последовательности. Такое ограничение делает данный подход не-пригодным для большинства задач, особенно для машинного перевода. Попробуйте перевести фразу «Сегодня хорошая погода» на французский — *Il fait beau aujourd'hui*. Для этого вам нужно будет научить модель предсказывать *Il* по слову «Сегодня», *Il fait* по «Сегодня хорошая» и т. д. — что просто невозможно.

Человек-переводчик сначала читает исходное предложение целиком и только потом приступает к переводу. Это особенно важно при работе с языками с совершенно разным порядком слов, например английским и японским. Именно так и поступают стандартные модели последовательностей.

Порядок действий при правильном подходе к обучению вида «последовательность в последовательность» следующий (рис. 11.13). Сначала нужно взять рекуррентную сеть (кодировщик), чтобы преобразовать всю исходную последовательность в один вектор (или набор векторов). Это может быть последний выход рекуррентной сети или конечные векторы внутреннего состояния. Затем данный вектор (или векторы) используется в качестве *начального состояния* другой рекуррентной сети (декодера), просматривающей элементы 0... $N$  целевой последовательности и пытающейся предсказать шаг  $N + 1$  в целевой последовательности.



**Рис. 11.13.** Рекуррентная модель типа «последовательность в последовательность»: рекуррентная сеть кодировщика создает вектор, кодирующий всю исходную последовательность, а далее этот вектор используется в качестве начального состояния рекуррентной сети декодера

Реализуем это решение в Keras, используя для реализации кодировщика и декодера слои GRU. Выбор GRU вместо LSTM немного упрощает задачу, поскольку GRU имеет только один вектор состояния, а LSTM — несколько. Начнем с кодировщика.

#### Листинг 11.28. Кодировщик на основе слоя GRU

```
from tensorflow import keras
from tensorflow.keras import layers

embed_dim = 256
latent_dim = 1024

source = keras.Input(shape=(None,), dtype="int64", name="english")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
encoded_source = layers.Bidirectional(
    layers.GRU(latent_dim), merge_mode="sum")(x)
```

Не забывайте про маску:  
это важно в данном решении

Исходное предложение на английском языке. Определение имени набора входных данных позволяет нам вызвать метод fit()  
модели для ее обучения с входным словарем

Результат кодирования исходного предложения — это последний выход двунаправленного слоя GRU

Далее добавим декодер — простой слой GRU, принимающий закодированное исходное предложение в качестве начального состояния. Также нам понадобится слой **Dense**, создающий для каждого шага вывода распределение вероятностей по испанскому словарю.

#### Листинг 11.29. Декодер на основе слоя GRU и полная модель

```

Целевое предложение на испанском                                Не забывайте про маски
→ past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
    x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target) ←
    decoder_gru = layers.GRU(latent_dim, return_sequences=True)
→ x = decoder_gru(x, initial_state=encoded_source)                  Предсказывает
    x = layers.Dropout(0.5)(x)                                         следующий токен
    target_next_step = layers.Dense(vocab_size, activation="softmax")(x) ←
    seq2seq_rnn = keras.Model([source, past_target], target_next_step) ←

Закодированное исходное                                      Полная модель: сопоставляет исходное
предложение служит начальным                                 предложение и целевое предложение с целевым
состоянием декодера GRU                                     предложением на один шаг в будущем

```

Во время обучения декодер принимает на входе всю целевую последовательность, но из-за пошагового характера рекуррентной сети просматривает только входные токены  $0 \dots N$ , чтобы предсказать токен  $N$  на выходе (который соответствует следующему токену в последовательности, потому что выходные данные смешены на один шаг). Это означает, что для предсказания будущего используется только информация из прошлого — так и должно быть, ведь в противном случае наша модель не работала бы на этапе прогнозирования и мы бы лишь обманули сами себя.

Приступим к обучению.

#### Листинг 11.30. Обучение рекуррентной модели типа «последовательность в последовательность»

```

seq2seq_rnn.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
seq2seq_rnn.fit(train_ds, epochs=15, validation_data=val_ds)

```

В качестве грубой оценки качества модели для мониторинга процесса обучения мы выбрали метрику точности на проверочном наборе данных. Точность достигла 64 % — то есть в среднем модель правильно предсказывает следующее слово в испанском предложении в 64 % случаев. Однако на практике для моделей машинного перевода точность предсказания следующего слова не лучшая метрика, в частности, потому, что она предполагает, что правильные целевые токены от 0 до  $N$  уже известны при прогнозировании токена  $N + 1$ . На самом деле во время перевода вы генерируете целевое предложение с нуля и не можете гарантировать, что ранее сгенерированные токены будут на 100 % правильными.

Для оценки реальных систем машинного перевода чаще всего используются баллы BLEU — показатель, рассматривающий сгенерированные последовательности целиком и, по-видимому, хорошо коррелирующий с человеческим восприятием качества перевода.

А теперь давайте опробуем нашу модель. Выберем несколько предложений из тестового набора и проверим, как модель их переведет. Прежде всего возьмем начальный токен "[start]" и передадим его в модель декодера вместе с закодированным английским исходным предложением. Далее мы получим предсказание следующего токена и будем повторно вводить его в декодер, выбирая новый целевой токен в каждой итерации, пока не дойдем до токена "[end]" или не достигнем максимальной длины предложения.

### Листинг 11.31. Перевод новых предложений с помощью кодировщика и декодера на основе рекуррентных сетей

```
Подготовка словаря для преобразования индекса
предсказываемого токена в строковый токен
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]" ← Начальный токен
    for i in range(max_decoded_sentence_length): ← Выборка следующего токена
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]": ←
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

При работе над этим листингом я добавил комментарии в виде блоков с правильными переводами на русский язык. Красные стрелки указывают на соответствующие места в коде.

- Подготовка словаря для преобразования индекса предсказываемого токена в строковый токен
- Начальный токен
- Выборка следующего токена
- Преобразование следующего предсказанного токена в строку и добавление ее в конец сгенерированного предложения
- Условие выхода: либо достигнута максимальная длина предложения, либо получен конечный токен

Обратите внимание: несмотря на всю свою простоту, данное решение весьма неэффективно — мы снова и снова обрабатываем исходное предложение и каждый раз генерируем целевое предложение, пробуя новое слово. В практическом приложении мы реализовали бы кодировщик и декодер в виде двух отдельных

моделей, и декодер выполнял бы только один шаг в каждой итерации выборки токена, повторно используя предыдущее внутреннее состояние.

Вот итог перевода. Наша модель показала неплохие результаты, хотя и допускает много ошибок.

**Листинг 11.32.** Некоторые результаты, сгенерированные рекуррентной моделью машинного перевода

```
Who is in this room?  
[start] quién está en esta habitación [end]  
-  
That doesn't sound too dangerous.  
[start] eso no es muy difícil [end]  
-  
No one will stop me.  
[start] nadie me va a hacer [end]  
-  
Tom is friendly.  
[start] tom es un buen [UNK] [end]
```

Эту простенькую модель можно улучшить, например взяв более глубокий стек рекуррентных слоев в кодировщике и в декодере (обратите внимание, что в таком случае управление состоянием декодера несколько усложнится). Мы могли бы использовать слои LSTM вместо GRU. И так далее. Однако в целом применение рекуррентных сетей к обучению «последовательность в последовательность» имеет несколько фундаментальных ограничений:

- представление исходной последовательности должно целиком храниться в векторе (векторах) состояния кодировщика, что существенно ограничивает размер и сложность переводимых предложений. Это как если бы человек полностью переводил предложение по памяти, не заглядывая в исходное предложение;
- рекуррентные сети плохо справляются с очень длинными последовательностями, потому что имеют склонность постепенно забывать прошлое — по достижении 100-го токена в любой последовательности в модели остается мало информации о начале данной последовательности. Следовательно, модели на основе RNN не могут удерживать длинный контекст, что может быть необходимо для перевода больших документов.

Эти ограничения привели к тому, что для решения задач преобразования последовательностей в последовательности в сообществе машинного обучения было отдано предпочтение архитектуре Transformer. Давайте рассмотрим ее более подробно.

### 11.5.3. Обучение типа «последовательность в последовательность» архитектуры Transformer

Обучение «последовательность в последовательность» — это задача, с которой архитектура Transformer справляется особенно хорошо. Механизм нейронного внимания позволяет моделям Transformer успешно обрабатывать более длинные и более сложные последовательности, чем в случае рекуррентных сетей.

Человек, переводящий тексты с английского на испанский, не будет читать английское предложение по одному слову, запоминать его значение, а затем генерировать предложение на испанском по одному слову за раз. Такой подход еще может сработать для коротких выражений из пяти слов, но вряд ли подойдет для целого абзаца. Вместо этого переводчик неоднократно будет просматривать исходное предложение и перевод и обращать внимание на разные слова в исходном тексте, записывая разные части перевода.

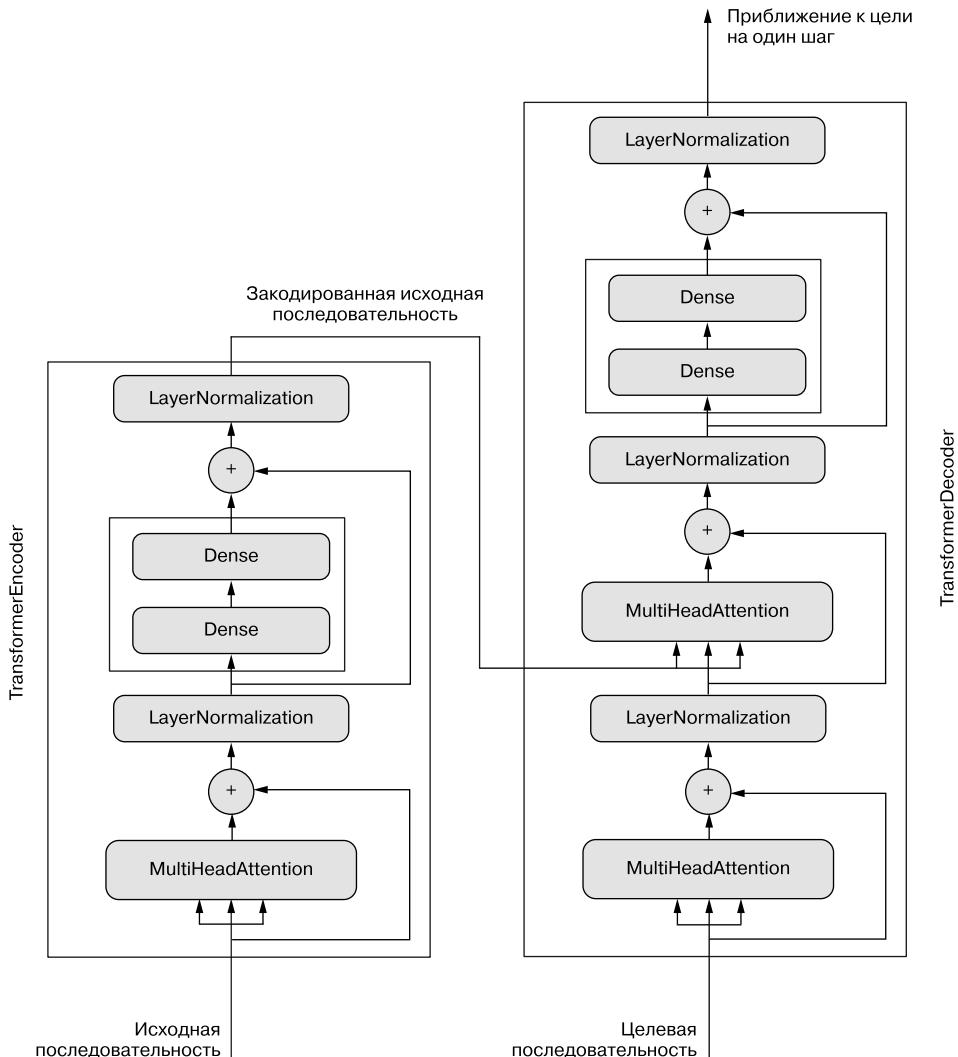
Именно этого позволяет достичь нейронное внимание и Transformer. Вы уже знакомы с кодировщиком Transformer, который использует внутреннее внимание для создания контекстно зависимых представлений для каждого слова во входной последовательности. Кодировщик Transformer в системе машинного перевода, естественно, будет читать исходную последовательность и создавать ее кодированное представление. Однако, в отличие от предыдущего кодировщика на основе рекуррентной сети, Transformer хранит закодированное представление в форме последовательности контекстно зависимых векторных представлений.

Вторая половина модели — *декодер Transformer*. Так же как декодер на основе рекуррентной сети, он читает токены  $0 \dots N$  в целевой последовательности и пытается предсказать токен  $N + 1$ . При этом он использует нейронное внимание, чтобы определить, какие токены в закодированном исходном предложении наиболее тесно связаны с целевым токеном, который он в настоящее время пытается предсказать, — возможно, этим он мало отличается от переводчика-человека. Вспомните модель «запрос — ключ — значение»: в декодере Transformer целевая последовательность служит запросом, используемым, чтобы уделить более пристальное внимание различным частям исходной последовательности (которая играет две роли: ключей и значений).

#### Декодер Transformer

На рис. 11.14 показана полная структура модели Transformer для обучения «последовательность в последовательность». Посмотрите, как устроен декодер: он очень похож на кодировщик Transformer, за исключением дополнительного блока внимания между блоком внутреннего внимания, который применяется к целевой последовательности, и полно связанными слоями в выходном блоке.

Давайте реализуем эту модель. Так же как в случае с `TransformerEncoder`, реализуем свой подкласс класса `Layer`. Прежде чем сосредоточиться на методе `call()`, в котором происходят все действия, определим конструктор класса, создающий необходимые слои.



**Рис. 11.14.** `TransformerDecoder` похож на `TransformerEncoder`, за исключением дополнительного блока внимания, где ключи и значения представляют исходную последовательность, закодированную с помощью `TransformerEncoder`. Вместе кодировщик и декодер образуют сквозную модель `Transformer`

**Листинг 11.33.** TransformerDecoder

```

class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),])
    )
    self.layernorm_1 = layers.LayerNormalization()
    self.layernorm_2 = layers.LayerNormalization()
    self.layernorm_3 = layers.LayerNormalization()
    self.supports_masking = True ←
def get_config(self):
    config = super().get_config()
    config.update({
        "embed_dim": self.embed_dim,
        "num_heads": self.num_heads,
        "dense_dim": self.dense_dim,
    })
    return config

```

Этот атрибут гарантирует, что слой будет распространять свою входную маску на свои выходные данные; маскировка в Keras должна включаться явно. Если передать маску слово, который не реализует метод `compute_mask()` и не поддерживает атрибут `supports_masking`, данная строка вызовет ошибку

Метод `call()` почти в точности воспроизводит диаграмму на рис. 11.14. Но есть еще одна деталь, которую мы должны учесть: *каузальное заполнение (causal padding)*. Оно совершенно необходимо для успеха обучения модели Transformer «последовательность в последовательность». В отличие от рекуррентных сетей, просматривающих входные данные шаг за шагом и потому имеющих доступ только к шагам  $0 \dots N$  при генерации выходного шага  $N$  (который является токеном  $N+1$  в целевой последовательности), декодер `TransformerDecoder` безразличен к порядку: он просматривает всю целевую последовательность сразу. Если позволить ему использовать всю входную последовательность, он бы просто научился копировать входной шаг  $N+1$  в точку  $N$  на выходе. В результате модель достигла бы идеальной точности на обучающих данных, но при прогнозировании была бы абсолютно бесполезна, раз на входе шаги больше  $N$  недоступны.

Решается проблема просто: нужно замаскировать верхнюю половину матрицы попарного внимания, чтобы модель не обращала внимания на информацию из будущего. При этом для создания целевого токена  $N+1$  следует использовать только информацию из токенов  $0 \dots N$  в целевой последовательности. Чтобы обеспечить это, добавим метод `get_causal_attention_mask(self, inputs)` в наш

класс `TransformerDecoder`, возвращающий маску внимания, которую можно передать слоям `MultiHeadAttention`.

**Листинг 11.34.** Метод в классе `TransformerDecoder`, генерирующий каузальную маску

```
def get_causal_attention_mask(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, sequence_length = input_shape[0], input_shape[1]
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    mask = tf.cast(i >= j, dtype="int32")
    mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1),
         tf.constant([1, 1], dtype=tf.int32)], axis=0)
    return tf.tile(mask, mult)
```

Сгенерировать матрицу с формой  
(длина\_последовательности,  
длина\_последовательности)  
с единицами в одной половине  
и с нулями в другой

Скопировать ее вдоль оси пакетов, чтобы  
получить матрицу с формой (размер\_пакета,  
длина\_последовательности, длина\_последовательности)

Теперь можно записать метод `call()`, реализующий прямой проход декодера.

**Листинг 11.35.** Прямой проход декодера `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs) ← Получить каузальную маску
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :, dtype="int32") ← Подготовить входную маску
        padding_mask = tf.minimum(padding_mask, causal_mask) ← Объединить две маски
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask) ← Передать каузальную маску в первый слой
                                         внимания, который реализует внутреннее
                                         внимание для целевой последовательности
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask, ← Передать объединенную маску во второй
                                         слой внимания, который связывает исходную
                                         и целевую последовательности
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

## Собираем все вместе: модель Transformer для машинного перевода

Сквозная модель Transformer — это модель, которую мы будем обучать. Она отображает исходную и целевую последовательности в целевую последовательность на один шаг в будущем. Сквозная модель объединяет части, созданные нами до сих пор: слои PositionalEmbedding, TransformerEncoder и TransformerDecoder. Обратите внимание, что TransformerEncoder и TransformerDecoder не зависят от формы входных данных, поэтому вы можете увеличить количество слоев, чтобы создать более мощный кодировщик или декодер. В нашем примере мы оставим по одному экземпляру каждого.

### Листинг 11.36. Сквозная модель Transformer

```

embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x) ← Кодирование исходной последовательности

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs) ← Кодирование целевой последовательности и объединение ее с закодированной исходной последовательностью
x = layers.Dropout(0.5)(x) ← Предсказание слова в каждой позиции в выходе
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x) ←
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)

```

Теперь можно обучить модель — мы достигли точности 67 %, намного превзойдя модель на основе GRU.

### Листинг 11.37. Обучение модели Transformer «последовательность в последовательность»

```

transformer.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
transformer.fit(train_ds, epochs=30, validation_data=val_ds)

```

В заключение попробуем применить нашу модель для перевода английских предложений из контрольного набора. Конфигурация идентична той, что использовалась для аprobации модели на основе рекуррентной сети.

**Листинг 11.38.** Перевод новых предложений с помощью модели Transformer

```

import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization(
            [decoded_sentence])[:, :-1]
        predictions = transformer(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))

```

Выборка следующего токена  
Условие выхода  
Преобразование следующего предсказанного токена в строку и ее добавление в конец сгенерированной последовательности

Субъективно модель Transformer работает значительно лучше модели на основе GRU. Это все еще очень простая модель, но уже лучше предыдущей.

**Листинг 11.39.** Некоторые результаты, сгенерированные моделью Transformer машинного перевода

```

This is a song I learned when I was a kid.
[start] esta es una canción que aprendí cuando era chico [end] ←
-
She can play the piano.
[start] ella puede tocar piano [end]
-
I'm not who you think I am.
[start] no soy la persona que tú crees que soy [end]
-
It may have rained a little last night.
[start] puede que llueva un poco el pasado [end]

```

Исходное предложение гендерно нейтральное, но в этом переводе предполагается, что говорящий — мужчина. Имейте в виду, что модели перевода часто делают необоснованные предположения о своих входных данных, что приводит к алгоритмической предвзятости. В худшем случае модель может воспроизводить отложившуюся в ее памяти информацию, которая не имеет ничего общего с данными, обрабатываемыми в конкретный момент

На этом мы завершаем главу об обработке естественного языка — вы только что прошли путь от самых основ до полноценной архитектуры Transformer, способной выполнять переводы с английского языка на испанский. Умение учить машины понимать язык — последняя сверхспособность, которую вы можете смело добавить в свою коллекцию.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Есть два вида моделей обработки естественного языка: *модели мешка слов*, обрабатывающие наборы слов или  $N$ -грамм без учета их порядка, и *модели последовательностей*, учитывающие порядок слов. Модель мешка слов состоит из слоев Dense; модель последовательности может быть рекуррентной сетью, одномерной сверточной сетью или архитектурой Transformer.
- В задачах классификации текста для выбора наилучшей применимой модели (модели мешка слов или модели последовательности) можно использовать отношение количества образцов в обучающих данных к среднему количеству слов в образце.
- *Векторные представления слов* — это векторные пространства, моделирующие семантические отношения между словами как отношения расстояния между векторами, представляющими эти слова.
- *Обучение «последовательность в последовательность»* — это универсальный и мощный подход к обучению, который можно применять для решения многих задач обработки естественного языка, включая машинный перевод. Модель «последовательность в последовательность» состоит из двух компонентов: кодировщика, обрабатывающего исходную последовательность, и декодера, пытающегося предсказать будущие токены в целевой последовательности, просматривая предшествующие токены, которые были получены кодировщиком из исходной последовательности.
- *Нейронное внимание* — это способ создания контекстно зависимых представлений слов и основа архитектуры Transformer.
- Архитектура Transformer, состоящая из TransformerEncoder и TransformerDecoder, дает превосходные результаты в задачах преобразования последовательностей. Первую половину — TransformerEncoder — можно также использовать для классификации текста или для любой другой задачи обработки естественного языка с одним входом.

# 19

## Генеративное глубокое обучение

### В этой главе

- ✓ Генерирование текста.
- ✓ 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 несколько лет недоверие исчезло. Летом 2015 года мы развлекались с алгоритмом Google DeepDream, превращавшим изображения в психodelическую мешанину из собачьих глаз и парейдотических артефактов; в 2016 году мы начали использовать приложения для смартфонов, превращающие фотографии 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 соревноваться с людьми, сценаристами, художниками и композиторами. Впрочем, замена человека 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 *искусственного интеллекта*.

Художественное творчество в значительной мере заключается в распознавании образов и технических навыках. Многим именно эта часть процесса кажется малопривлекательной, 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, что 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 примеси ремесла.

Янис Ксенакис, пионер электронной и алгоритмической музыки, прекрасно выразил ту же идею в 1960-х годах 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 испытать все инструментальные комбинации, от одиночных инструментов до крупных оркестров. С помощью электронных компьютеров композитор может стать кем-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».*

<sup>1</sup> Xenakis I. Musiques formelles: nouveaux principes formels de composition musicale // La Revue musicale, 1963 Nos. 253–254.

В этой главе мы с разных сторон рассмотрим потенциал глубокого обучения для расширения творческих возможностей. Мы познакомимся с приемами генерирования последовательностей данных (которые можно использовать для создания текста или музыки), алгоритмом DeepDream и методами 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.

## 12.1. ГЕНЕРИРОВАНИЕ ТЕКСТА

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.

### 12.1.1. Краткая история генеративного глубокого обучения для генерирования последовательностей

В конце 2014 года даже в сообществе машинного обучения мало кто был знаком с аббревиатурой LSTM. 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. Этот новый алгоритм первое время использовался для генерации текстов символ за символом.

В 2002 году Дуглас Эк, а затем и исследователи в швейцарской лаборатории имени Шмидхубера 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 осязаемый результат.

В конце 2000-х — начале 2010-х годов Алекс Грэйвз 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, именно тогда данное конкретное применение нейронных сетей 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 открывающихся возможностей.

Затем, примерно в 2017–2018 годах появилась архитектура Transformer, использовавшая рекуррентные нейронные сети не только для задач обработки естественного языка, но и для моделей генерирования последовательностей, в частности *моделирования языка* (генерации текста на уровне слов). Самым известным примером генеративной модели с архитектурой Transformer является, пожалуй, модель GPT-3 с 175 миллиардами параметров, обученная стартапом OpenAI на поразительно большом текстовом корпусе, который включал множество книг, доступных в цифровом формате, статьи в «Википедии» и значительный объем текста, полученного сканированием интернета. В 2020 году модель GPT-3 даже попала в заголовки газет благодаря своей способности генерировать правдоподобно выглядящие абзацы текста практически на любую тему — такое ее мастерство вызвало кратковременную волну ажиотажа, достойную самого жаркого лета ИИ.

### 12.1.2. Как генерируются последовательности данных

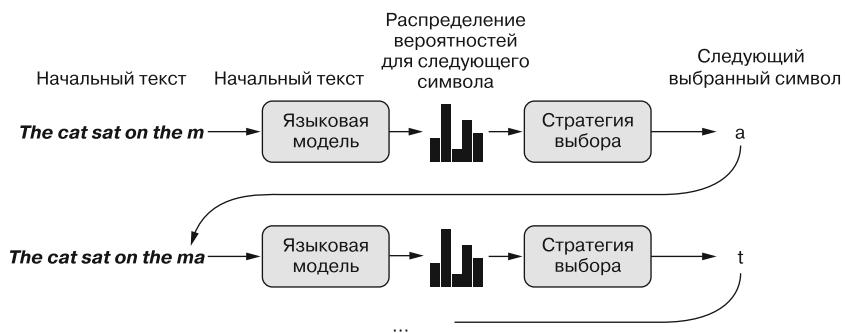
Универсальный способ генерации последовательностей данных с применением методов глубокого обучения заключается в обучении модели (обычно рекуррентной или с архитектурой Transformer) для прогнозирования следующего токена или следующих нескольких токенов в последовательности, опираясь

---

<sup>1</sup> Graves A. Generating Sequences With Recurrent Neural Networks // arXiv, 2013, <https://arxiv.org/abs/1308.0850>.

на предыдущие токены. Например, для входной последовательности *The cat is on the* модель обучается предсказывать *mat* — следующее целевое слово. 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. 12.1). Этот цикл позволяет генерировать последовательности произвольной длины, отражающие структуру данных, на которых обучалась модель: последовательности, которые выглядят *почти* как предложения, написанные человеком.



**Рис. 12.1.** Процесс пословной генерации текста с использованием языковой модели

### 12.1.3. Важность стратегии выбора

Для генерации текста важную роль играет алгоритм выбора следующего токена. Простейшее 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 слово имеет вероятность 0,3 стать следующим в предложении, согласно модели мы выберем его в 30 % случаев. Обратите внимание, 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 следующее слово выбирается из равномерно распределенных вероятностей и каждое слово одинаково вероятно. 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 потому 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.

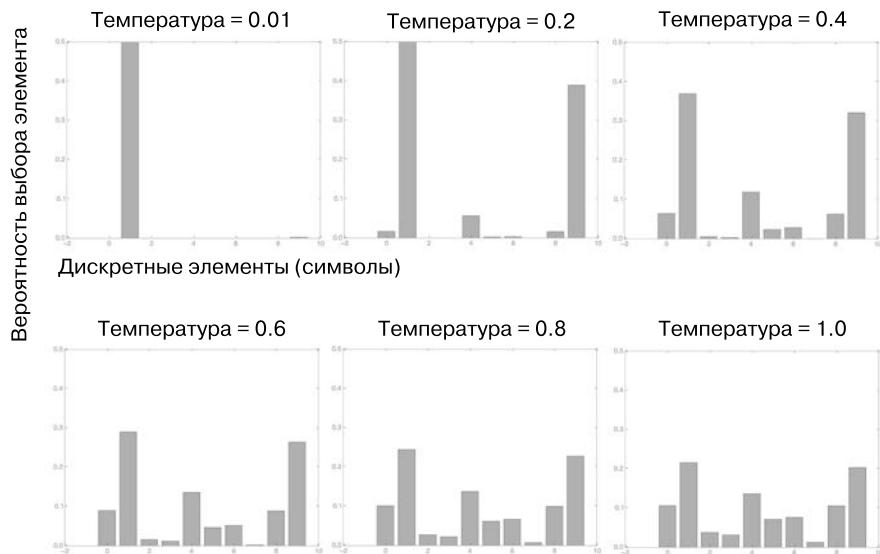
**Листинг 12.1.** Взвешивание распределения вероятностей с учетом значения температуры

```
original_distribution — это одномерный
массив NumPy значений вероятностей, сумма
которых должна быть равна 1; temperature —
это коэффициент, определяющий уровень
энтропии выходного распределения
```

```
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)
```

Возвращает 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 структуризованными будут генерируемые данные. Чем меньше температура, тем меньше будет величина случайной составляющей и тем более предсказуемыми будут генерируемые данные (рис. 12.2).



**Рис. 12.2.** Разные результаты взвешивания одного и того же распределения вероятностей. Низкая температура = высокая предсказуемость, высокая температура = более случайный результат

### 12.1.4. Реализация генерации текста в Keras

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 продолжим использовать набор данных IMDB с отзывами о фильмах из предыдущей главы и попробуем обучить модель генерировать новые отзывы к никогда не снимавшимся фильмам. В результате обучения у нас получится языковая модель, обладающая специфическими особенностями, характерными для отзывов о фильмах, а не обобщенная модель английского языка.

#### Подготовка данных

Так же как в предыдущей главе, загрузим и распакуем набор данных IMDB с отзывами о фильмах.

#### Листинг 12.2. Загрузка и распаковка набора данных IMDB с отзывами о фильмах

```
!wget https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

Вы уже знакомы со структурой данных: у нас есть папка с именем `aclImdb`, содержащая две подпапки: одна с негативными отзывами о фильмах и одна — с положительными. Каждый отзыв хранится в отдельном текстовом файле. Вызовем `text_dataset_from_directory` с параметром `label_mode=None`, чтобы создать объект набора данных, читающий эти файлы и возвращающий их содержимое.

#### Листинг 12.3. Создание набора данных из текстовых файлов (один файл = один образец)

```
Удаление HTML-тегов <br />, встречающихся во многих обзорах.
Это действие не повлияет на качество классификации текста,
но нам не хотелось бы генерировать теги <br /> в примере!
import tensorflow as tf
from tensorflow import keras
dataset = keras.utils.text_dataset_from_directory(
    directory="aclImdb", label_mode=None, batch_size=256)
dataset = dataset.map(lambda x: tf.strings.regex_replace(x, "<br />", " "))
```

Теперь используем слой `TextVectorization` для составления словаря, с которым будем работать. Извлечем из каждого отзыва только `sequence_length` первых слов: слой `TextVectorization` обрежет все остальное при векторизации текста.

**Листинг 12.4.** Подготовка слоя TextVectorization

```
from tensorflow.keras.layers import TextVectorization

sequence_length = 100
vocab_size = 15000
text_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
text_vectorization.adapt(dataset)
```

Давайте используем этот слой, чтобы создать набор данных для языковой модели, в котором входные образцы являются векторизованными фрагментами текста, а соответствующие цели — теми же фрагментами, но смещенными на одно слово.

**Листинг 12.5.** Настройка набора данных для языковой модели

```
def prepare_lm_dataset(text_batch):
    vectorized_sequences = text_vectorization(text_batch)
    x = vectorized_sequences[:, :-1]
    y = vectorized_sequences[:, 1:]
    return x, y
lm_dataset = dataset.map(prepare_lm_dataset, num_parallel_calls=4)
```

### Модель «последовательность в последовательность» на основе архитектуры Transformer

Наша цель — обучить модель прогнозировать распределение вероятностей для выбора следующего слова в предложении с учетом некоторого количества начальных слов. После обучения модели мы дадим ей подсказку, затем выберем следующее слово, добавим его в конец подсказки и снова передадим модели. И будем повторять это действие до тех пор, пока не сгенерируем короткий абзац.

По аналогии с прогнозированием температуры в главе 10 можно обучить модель, которая принимает на входе последовательность из  $N$  слов и просто

предсказывает слово  $N + 1$ . Однако этот подход имеет некоторые проблемы в контексте генерации последовательностей.

Во-первых, такая модель сможет делать прогнозы, только когда доступно  $N$  слов, но было бы полезно иметь возможность начать прогнозирование с количеством слов меньше  $N$ . В противном случае мы будем вынуждены брать только относительно длинные подсказки (в нашей реализации  $N = 100$  слов). В главе 10 мы не были ограничены таким требованием.

Во-вторых, многие из обучающих последовательностей будут в значительной степени перекрываться. Возьмем для примера  $N = 4$  и текст «Законченное предложение должно содержать как минимум три члена: подлежащее, сказуемое и дополнение» для создания следующих обучающих последовательностей:

- «Законченное предложение должно содержать»;
- «предложение должно содержать как»;
- «должно содержать как минимум»;
- и т. д. до последовательности «подлежащее, сказуемое и дополнение».

Модели, интерпретирующей каждую такую последовательность как независимый образец, пришлось бы выполнять много лишней работы, повторно кодируя подпоследовательности, которые она уже видела раньше. В главе 10 это не было большой проблемой, ведь у нас изначально не было такого количества обучающих образцов и нам нужно было протестировать полно связные и сверточные модели, для которых переделка работы каждый раз — единственный возможный вариант. Мы могли бы попытаться решить проблему избыточности, выбирая подпоследовательности с некоторым *шагом* — пропуская несколько слов между двумя последовательными образцами. Но это уменьшило бы количество обучающих образцов и лишь частично решило бы проблему.

Чтобы решить обе проблемы, используем модель «*последовательность в последовательность*»: мы будем передавать в модель последовательности из  $N$  слов (индексированных от 0 до  $N$ ) и предсказывать последовательность, смешенную на единицу (от 1 до  $N + 1$ ). При этом будем использовать каузальную маскировку, дабы убедиться, что для любого  $i$  модель будет брать только слова от 0 до  $i$  при предсказании слова  $i + 1$ . Таким образом, мы обучаем модель одновременно решать  $N$  во многом перекрывающихся, но все же разных задач: предсказание следующих слов по последовательности из  $1 \leq i \leq N$  предыдущих слов (рис. 12.3). На этапе генерации, даже если вы предложите модели только одно слово, она сможет дать распределение вероятностей для следующих возможных слов.

Предсказание следующего слова	the cat sat on the → mat
	the → cat sat on the mat
	the cat → sat on the mat
	the cat sat → on the mat
	the cat sat on → the mat
	the cat sat on the → mat

**Рис. 12.3.** По сравнению с простым предсказанием следующего слова, при моделировании «последовательность в последовательность» оптимизируется решение сразу нескольких задач предсказания

Обратите внимание, что мы могли бы использовать аналогичную модель «последовательность в последовательность» для решения задачи прогнозирования температуры в главе 10: научить модель на основе последовательности из 120 почасовых точек генерировать последовательность из 120 температур со смещением на 24 часа в будущем. Такая модель решала бы не только исходную задачу, но и 119 связанных задач прогнозирования температуры за 24 часа с учетом  $1 \leq i < 120$  предыдущих почасовых точек данных.

Попробовав обучить рекуррентную сеть из главы 10 в конфигурации «последовательность в последовательность», вы получили бы похожие, но постепенно ухудшающиеся результаты, потому что ограничение, требующее решения той же моделью дополнительных 119 задач, немного мешает работе над задачей, которая представляет основной интерес.

В предыдущей главе вы познакомились с обобщенным подходом к обучению типа «последовательность в последовательность», при котором исходная последовательность передается на вход кодировщика, а полученная от него за-кодированная последовательность вместе с целевой идет в декодер, который пытается предсказать ту же самую целевую последовательность со смещением на один шаг. Когда выполняется генерация текста, исходная последовательность отсутствует: модель просто пытается предсказать следующие токены в целевой последовательности, учитывая предыдущие, что можно сделать, используя только декодер. И благодаря каузальному дополнению декодер будет смотреть только на слова  $0 \dots N$ , чтобы предсказать слово  $N + 1$ .

Давайте реализуем нашу модель. Для этого повторно используем строительные блоки, созданные в главе 11: `PositionalEmbedding` и `TransformerDecoder`.

**Листинг 12.6.** Простая языковая модель на основе архитектуры Transformer

```
from tensorflow.keras import layers
embed_dim = 256
latent_dim = 2048
num_heads = 2

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="sparse_categorical_crossentropy", optimizer="rmsprop")
```

Значения softmax по возможным словам из словаря, рассчитанные для каждого временного шага выходной последовательности

### 12.1.5. Обратный вызов для генерации текста с разными значениями температуры

Для генерации текста с использованием различных температур после каждой эпохи мы применим обратный вызов. Это позволит увидеть, как меняется сгенерированный текст по мере схождения модели, а также как на стратегию выбора влияет температура. Чтобы начать генерацию текста, используем подсказку *this movie* («этот фильм»): все наши сгенерированные тексты будут начинаться с этого словосочетания.

**Листинг 12.7.** Обратный вызов для генерации текста

```
import numpy as np
```

Словарь, отображающий индексы слов в их строковые представления, для декодирования текста

```
tokens_index = dict(enumerate(text_vectorization.get_vocabulary()))
```

←

```
def sample_next(predictions, temperature=1.0):
    predictions = np.asarray(predictions).astype("float64")
    predictions = np.log(predictions) / temperature
    exp_preds = np.exp(predictions)
    predictions = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, predictions, 1)
    return np.argmax(probas)
```

← Реализует выбор из распределения вероятностей с учетом значения температуры

```
class TextGenerator(keras.callbacks.Callback):
    def __init__(self,
                 prompt,
                 generate_length,
                 model_input_length,
                 temperatures=(1.,),
                 print_freq=1):
        self.prompt = prompt
```

← Подсказка для начала генерирования текста

← Количество генерируемых слов

← Диапазон температур для выбора

```

self.generate_length = generate_length
self.model_input_length = model_input_length
self.temperatures = temperatures
self.print_freq = print_freq

def on_epoch_end(self, epoch, logs=None):
    if (epoch + 1) % self.print_freq != 0:
        return
    for temperature in self.temperatures:
        print("== Generating with temperature", temperature)
        sentence = self.prompt
        for i in range(self.generate_length):
            tokenized_sentence = text_vectorization([sentence])
            predictions = self.model(tokenized_sentence)
            next_token = sample_next(predictions[0, i, :])
            sampled_token = tokens_index[next_token]
            sentence += " " + sampled_token
            print(sentence)
    prompt = "This movie"
    text_gen_callback = TextGenerator(
        prompt,
        generate_length=50,
        model_input_length=sequence_length,
        temperatures=(0.2, 0.5, 0.7, 1., 1.5))

```

**Генерирование текста начинается с подсказки**

**Получаем прогнозы для последнего временного шага и используем их для выбора нового слова**

**Добавляем новое слово в конец текущей последовательности и повторяем**

**Для демонстрации влияния температуры на генерацию текста используем несколько разных температур**

А теперь обучим модель.

#### Листинг 12.8. Обучение языковой модели

```
model.fit(lm_dataset, epochs=200, callbacks=[text_gen_callback])
```

Вот несколько примеров, демонстрирующих возможности модели после 200 эпох обучения. Обратите внимание, что знаки препинания не являются частью словаря, поэтому ни один из сгенерированных текстов не содержит знаков препинания.

- Со значением `temperature=0.2`:
  - *this movie is a [UNK] of the original movie and the first half hour of the movie is pretty good but it is a very good movie it is a good movie for the time period;*
  - *this movie is a [UNK] of the movie it is a movie that is so bad that it is a [UNK] movie it is a movie that is so bad that it makes you laugh and cry at the same time it is not a movie i dont think ive ever seen.*
- Со значением `temperature=0.5`:
  - *this movie is a [UNK] of the best genre movies of all time and it is not a good movie it is the only good thing about this movie i have seen it for the first time and i still remember it being a [UNK] movie i saw a lot of years;*

- *this movie is a waste of time and money i have to say that this movie was a complete waste of time i was surprised to see that the movie was made up of a good movie and the movie was not very good but it was a waste of time and.*
- Со значением **temperature=0.7**:
  - *this movie is fun to watch and it is really funny to watch all the characters are extremely hilarious also the cat is a bit like a [UNK] [UNK] and a hat [UNK] the rules of the movie can be told in another scene saves it from being in the back of;*
  - *this movie is about [UNK] and a couple of young people up on a small boat in the middle of nowhere one might find themselves being exposed to a [UNK] dentist they are killed by [UNK] i was a huge fan of the book and i havent seen the original so it.*
- Со значением **temperature=1.0**:
  - *this movie was entertaining i felt the plot line was loud and touching but on a whole watch a stark contrast to the artistic of the original we watched the original version of england however whereas arc was a bit of a little too ordinary the [UNK] were the present parent [UNK];*
  - *this movie was a masterpiece away from the storyline but this movie was simply exciting and frustrating it really entertains friends like this the actors in this movie try to go straight from the sub thats image and they make it a really good tv show.*
- Со значением **temperature=1.5**:
  - *this movie was possibly the worst film about that 80 women its as weird insightful actors like barker movies but in great buddies yes no decorated shield even [UNK] land dinosaur ralph ian was must make a play happened falls after miscast [UNK] bach not really not wrestlemania seriously sam didnt exist;*
  - *this movie could be so unbelievably lucas himself bringing our country wildly funny things has is for the garish serious and strong performances colin writing more detailed dominated but before and that images gears burning the plate patriotism we you expected dyan bosses devotion to must do your own duty and another.*

Как видите, при низком значении температуры генерируется довольно скучный и повторяющийся текст, и иногда процесс генерирования просто зацикливается. При высоком значении температуры формируется более интересный текст, неожиданный и даже творческий, но внутренняя структура начинает разрушаться и текст выглядит как случайный набор слов. Без сомнения, 0,7 – 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, — вывод такой модели, как GPT-3, наглядно демонстрирует, чего можно ожидать от языковых моделей (GPT-3 — это фактически та же модель, которую мы обучили в этом примере, но с большим количеством декодеров Transformer и большим объемом обучающих данных, на которых она настроена). Однако не думайте, что когда-нибудь вам удастся сгенерировать осмысленный текст, разве только по чистой случайности и согласно вашей личной интерпретации: вы лишь выбираете образцы данных из статистической модели, в которой слова следуют за словами.

Естественный язык многообразен — это и канал общения, и способ воздействия на мир, и социальная смазка, и способ формулировать, хранить и извлекать собственные мысли... Из этих разных применений языка берет начало его смысл. «Языковая модель» глубокого обучения, несмотря на свое название, на самом деле не отражает ни одного из данных фундаментальных аспектов языка. Она не умеет общаться (ей не о чем и не с кем общаться), она не способна воздействовать на мир (у нее нет намерений и точек приложения воздействий), она не может быть социальной, и у нее нет мыслей, которые можно было бы выразить с помощью слов. Язык — это операционная система разума, поэтому, чтобы язык имел смысл, ему нужен разум, который бы его использовал.

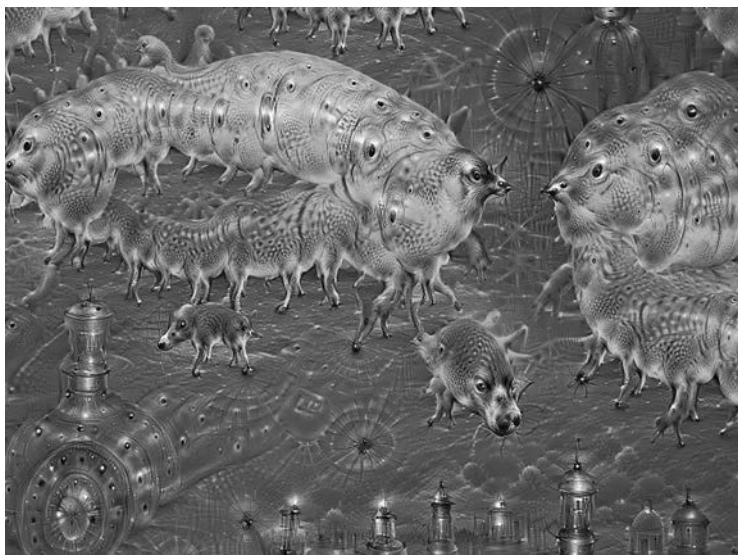
Языковая модель просто фиксирует статистическую структуру наблюдаемых артефактов — книг, онлайн-обзоров фильмов, твитов, — которые мы генерируем, применяя язык в своей жизни. Наличие статистической структуры в этих артефактах является лишь побочным эффектом того, как люди реализуют язык. Проведите мысленный 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.

### 12.1.6. 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.

## 12.2. 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. 12.4), 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. 12.4.** Пример изображения, созданного с помощью DeepDream

Алгоритм DeepDream почти идентичен методу визуализации фильтров сверточных сетей, представленному в главе 9, и состоит из сверточной сети, действующей в обратном направлении: выполняет градиентное восхождение по входным данным, максимизируя активацию определенного фильтра в 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> Mordvintsev A., Olah C., Tyka M. 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.

### 12.2.1. Реализация DeepDream в Keras

Для начала получим изображение для создания изобразительных фантазий. Мы будем использовать вид на сухое побережье Северной Калифорнии зимой (рис. 12.5).

**Листинг 12.9.** Получение изображения для экспериментов

```
from tensorflow import keras
import matplotlib.pyplot as plt

base_image_path = keras.utils.get_file(
    "coast.jpg", origin="https://img-datasets.s3.amazonaws.com/coast.jpg")

plt.axis("off")
plt.imshow(keras.utils.load_img(base_image_path))
```



**Рис. 12.5.** Изображение для экспериментов

Нам также понадобится предварительно обученная сверточная сеть. В Keras имеется несколько таких сетей: VGG16, VGG19, Xception, ResNet50 и т. д., и все они доступны с весами, предварительно обученными на наборе данных ImageNet. 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; на практике она известна красиво выглядящими картинками DeepDreams, поэтому мы возьмем модель Inception V3, входящую в состав Keras.

#### Листинг 12.10. Создание экземпляра предварительно обученной модели Inception V3

```
from tensorflow.keras.applications import inception_v3
model = inception_v3.InceptionV3(weights="imagenet", include_top=False)
```

Мы используем предварительно обученную сверточную сеть для создания модели извлечения признаков, которая будет возвращать активации различных промежуточных слоев, перечисленных в листинге 12.11. Для каждого слоя мы получим скалярную оценку, взвешивающую вклад слоя в потери, которые мы будем максимизировать в процессе градиентного восхождения. Если вам нужен полный список имен слоев, чтобы попробовать разные их комбинации, просто используйте `model.summary()`.

#### Листинг 12.11. Определение вклада каждого слоя в потери DeepDream

```
layer_settings = {
    "mixed4": 1.0,
    "mixed5": 1.5,
    "mixed6": 2.0,
    "mixed7": 2.5,
}
outputs_dict = dict([
    (layer.name, layer.output)
    for layer in [model.get_layer(name)
                  for name in layer_settings.keys()])
])
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

Слой, для которых мы постараемся максимизировать активации, и их веса в общей сумме потерь. Эти параметры можно настраивать и получать новые визуальные эффекты

Символические выходы каждого слоя

Модель, возвращающая значения активаций для каждого целевого слоя (в форме словаря)

Теперь вычислим *потери*: величину, которую мы будем максимизировать в процессе градиентного восхождения. В главе 9, 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 12.12. Определение потерь для DeepDream

```
def compute_loss(input_image):
    features = feature_extractor(input_image)
    loss = tf.zeros(shape=())
    for name in features.keys():
        coeff = layer_settings[name]
        activation = features[name]
        loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :]))
    return loss
```

Извлечение активаций  
Инициализация значений потерь нулями  
Чтобы избежать краевых эффектов, из вычисления потерь исключаются пиксели, находящиеся на краях изображения

Теперь настроим процесс градиентного восхождения, который будет применяться к каждой октаве. Вы наверняка заметите, что здесь используется тот же прием, что и для визуализации фильтров в главе 9! Алгоритм DeepDream — это просто форма визуализации фильтра в нескольких масштабах.

### Листинг 12.13. Процесс градиентного восхождения

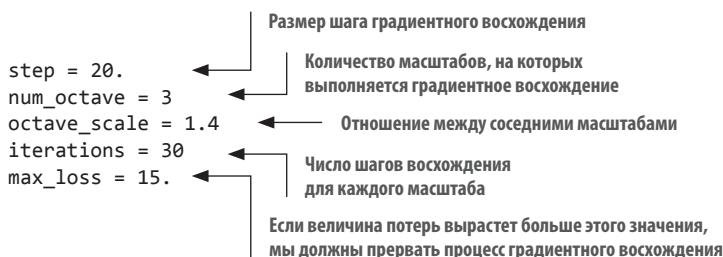
```
import tensorflow as tf
@tf.function
def gradient_ascent_step(image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = compute_loss(image)
        grads = tape.gradient(loss, image)
        grads = tf.math.l2_normalize(grads)
        image += learning_rate * grads
    return loss, image
```

```
def gradient_ascent_loop(image, iterations, learning_rate, max_loss=None):
    for i in range(iterations):
        loss, image = gradient_ascent_step(image, learning_rate)
        if max_loss is not None and loss > max_loss:
            break
        print(f"... Loss value at step {i}: {loss:.2f}")
    return image
```

Чтобы ускорить шаг обучения, скомпилируем его как tf.function  
Вычислить градиенты потерь DeepDream для текущего изображения  
Нормализовать градиенты (этот трюк уже использовался в главе 9)  
Запускает градиентное восхождение для данного масштаба изображения (октавы)  
Обновить изображения в цикле, чтобы увеличить потери DeepDream  
Прервать цикл, если потери превысили определенный порог (чрезмерная оптимизация приведет к появлению нежелательных артефактов на изображении)

Наконец добавим внешний цикл алгоритма DeepDream. Сначала определим список *масштабов* (также называемых *октавами*), в которых будут обрабатываться изображения. Мы будем обрабатывать наше изображение с тремя разными октавами. Для каждой последующей октавы, от самой маленькой до самой большой, мы будем выполнять 20 шагов градиентного восхождения, вызывая `gradient_ascent_loop()`, чтобы максимизировать потери, которые мы определили выше. Каждая следующая октава будет крупнее предыдущей на 40% (в 1,4 раза): мы начнем с обработки маленького изображения и затем постепенно будем увеличивать его (рис. 12.6).

Параметры этого процесса определяются в следующем фрагменте кода. Настраивая их, вы сможете добиться новых эффектов!



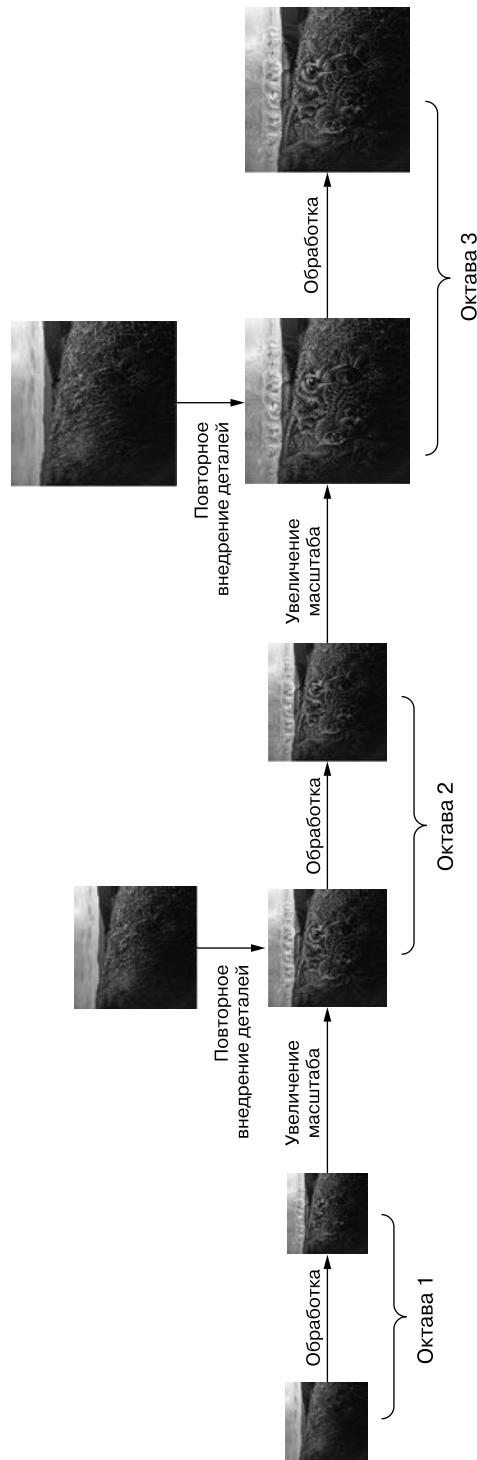
Нам также понадобится пара вспомогательных функций для загрузки и сохранения изображения.

**Листинг 12.14.** Вспомогательные функции для обработки изображений

```
import numpy as np

def preprocess_image(image_path): ← Открывает изображение,
    img = keras.utils.load_img(image_path) | изменяет его размер и преобразует
    img = keras.utils.img_to_array(img) | в соответствующий массив
    img = np.expand_dims(img, axis=0)
    img = keras.applications.inception_v3.preprocess_input(img)
    return img

def deprocess_image(img): ← Вспомогательная функция
    img = img.reshape((img.shape[1], img.shape[2], 3)) | для преобразования массива NumPy
    img /= 2.0 | в допустимое изображение
    img += 0.5 | Отмена операций,
    img *= 255. | выполненных
    img *= 255. | моделью Inception v3
    img = np.clip(img, 0, 255).astype("uint8") ← Преобразование в тип uint8
    return img | и усечение до допустимого
                | диапазона [0, 255]
```



**Рис. 12.6.** Процесс DeepDream: последовательные масштабы пространственной обработки (октавы) и повторное внедрение деталей при увеличении масштаба

А теперь перейдем к внешнему циклу. 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 размером  $S$  и большое — 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 12.15. Выполнение градиентного восхождения через последовательность октав

```
Выполнение градиентного восхождения
с изменением изображения

original_img = preprocess_image(base_image_path) ←
original_shape = original_img.shape[1:3]           | Загрузка
successive_shapes = [original_shape]               | базового
for i in range(1, num_octave):                    | изображения
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape]) ←
    successive_shapes.append(shape)
successive_shapes = successive_shapes[::-1]         | Вычисление целевой
                                                        | формы изображения
                                                        | для разных октав

shrunk_original_img = tf.image.resize(original_img, successive_shapes[0]) ←
img = tf.identity(original_img) ←                   | Создание копии изображения
                                                        | (оригинал нам еще понадобится)
for i, shape in enumerate(successive_shapes):       ←
    print(f"Processing octave {i} with shape {shape}") ←
    img = tf.image.resize(img, shape) ←             | Выполнение
                                                        | обхода разных
                                                        | октав
    img = gradient_ascent_loop(                     ←
        img, iterations=iterations, learning_rate=step, max_loss=max_loss
    )
→ upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape) ←
same_size_original = tf.image.resize(original_img, shape) ←
lost_detail = same_size_original - upscaled_shrunk_original_img ←
                                                        | Увеличение изображения
                                                        | Повторное внедрение деталей
                                                        | Сохранение результата
img += lost_detail
shrunk_original_img = tf.image.resize(original_img, shape)

keras.utils.save_img("dream.png", deprocess_image(img.numpy())) ←

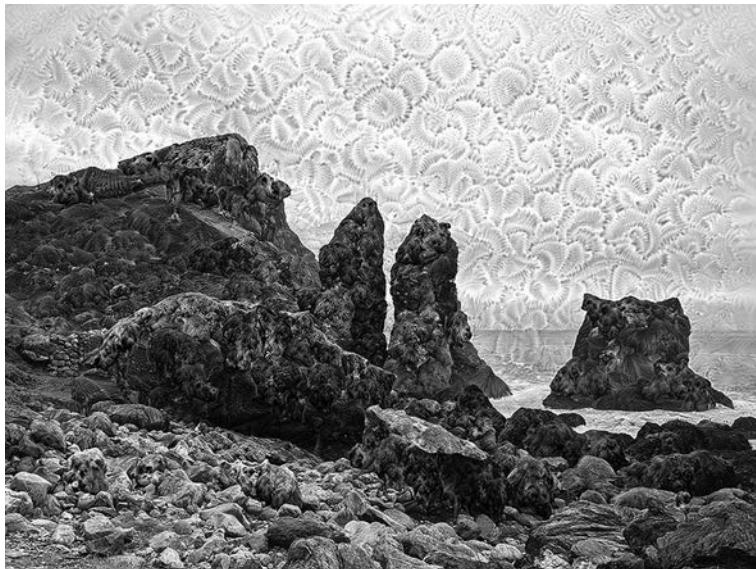
Увеличение уменьшенной версии
исходного изображения: в результате
появится эффект мозаичности
Разница между двумя изображениями —
это детали, утерянные в результате
масштабирования
Вычисление улучшенной
версии исходного изображения
заданного размера
```

## ПРИМЕЧАНИЕ

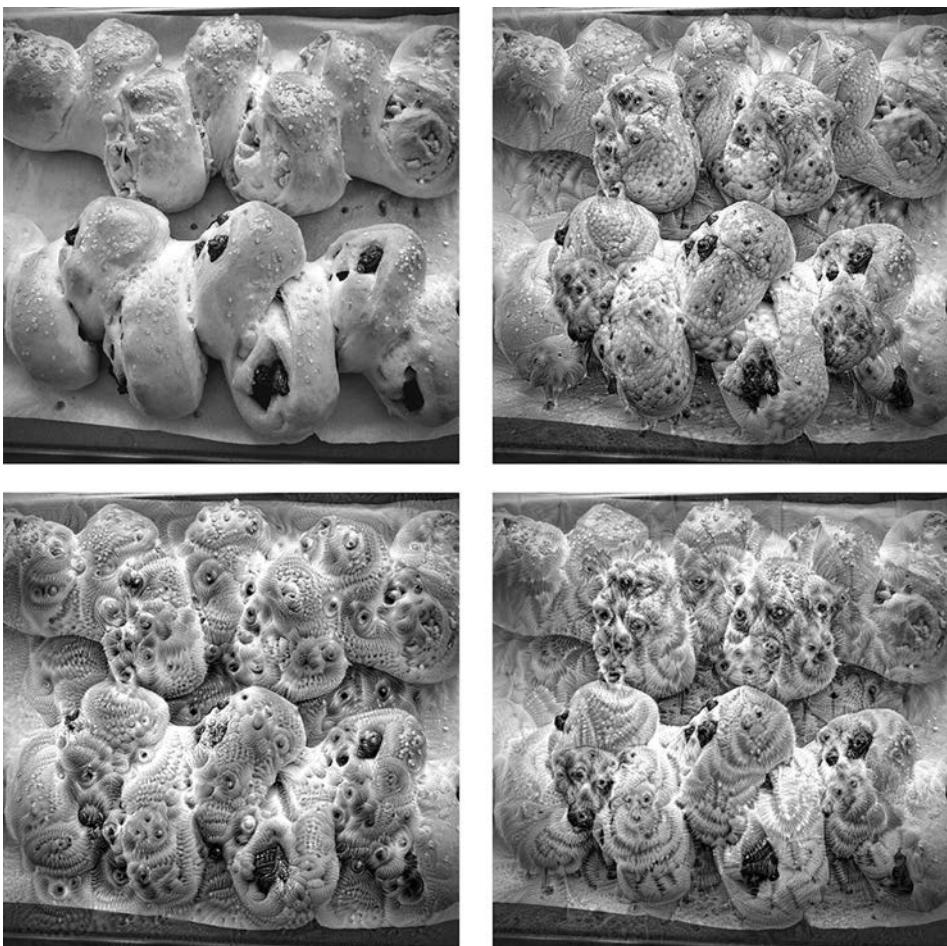
Оригинальная сеть Inception V3 была обучена распознавать образы на изображениях размером  $299 \times 299$ , поэтому, с учетом того что процесс выполняет уменьшение масштаба изображений с разумным коэффициентом, реализация DeepDream производит лучшие результаты для изображений размерами между  $300 \times 300$  и  $400 \times 400$ . Тем не менее вы можете использовать тот же код для обработки изображений любого размера с любым соотношением сторон.

На GPU весь код выполняется всего несколько секунд. На рис. 12.7 показан результат обработки выбранного нами изображения алгоритмом DeepDream.

Настоятельно рекомендую 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_settings`, чтобы быстро оценить множество разных комбинаций слоев. На рис. 12.8 показан диапазон результатов, полученных с использованием разных конфигураций слоев из изображения с домашними булочками.



**Рис. 12.7.** Результат обработки изображения реализацией алгоритма DeepDream



**Рис. 12.8.** Применение разных конфигураций DeepDream к изображению

### 12.2.2. 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.

## 12.3. НЕЙРОННАЯ ПЕРЕДАЧА СТИЛЯ

Кроме DeepDream, существует еще одна важная разработка в области изменения изображений с использованием глубокого обучения — *нейронная передача стиля*, реализованная Леоном Гатисом 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. 12.9.

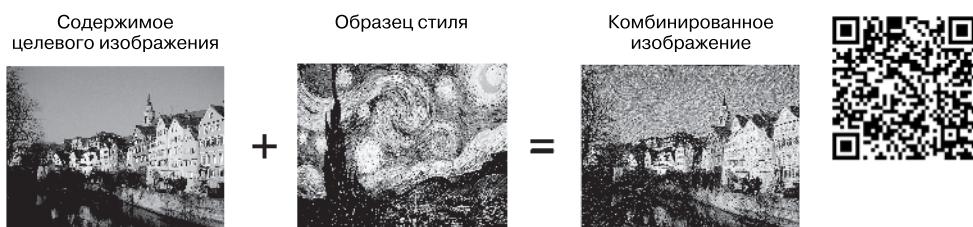


Рис. 12.9. Пример передачи стиля

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. 12.9 соответствуют стилю (в качестве образца использована картина Винсента Ван Гога «Звездная ночь»), а здания на фотографии, сделанной фотографом Тюбингеном, — это содержимое.

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 потому 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

<sup>1</sup> Gatys L. A., Ecker A. S., Bethge M. A Neural Algorithm of Artistic Style // arXiv, 2015, <https://arxiv.org/abs/1508.06576>.

изображения-образца. Определив математически *содержимое и стиль*, соответствующую функцию потерь для минимизации можно обозначить так:

```
loss = (distance(style(reference_image) - style(combination_image)) +  
       distance(content(original_image) - content(combination_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(combination_image)` приближается к `style(reference_image)`, а `content(combination_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.

### 12.3.1. 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 обеспечивают 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.

### 12.3.2. 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 использует *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*: изображение-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.

### 12.3.3. Нейронная передача стиля в Keras

Нейронную передачу стиля можно реализовать с использованием любой обученной сверточной сети. Здесь мы возьмем сеть VGG19, которую использовали Гатис с коллегами. VGG19 — это упрощенный вариант сети VGG16, представленной в главе 9, 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 400 пикселей.

**Листинг 12.16.** Получение стиля и содержимого изображений

```
from tensorflow import keras
base_image_path = keras.utils.get_file( ← Путь к изображению, которое
    "sf.jpg", origin="https://img-datasets.s3.amazonaws.com/sf.jpg") будт трансформироваться
style_reference_image_path = keras.utils.get_file( ← Путь к изображению
    "starry_night.jpg", с образом стиля
    origin="https://img-datasets.s3.amazonaws.com/starry_night.jpg")
original_width, original_height = keras.utils.load_img(base_image_path).size
img_height = 400
img_width = round(original_width * img_height / original_height) | Размеры генерируемого
                                                               изображения
```

Изображение, которое будет служить нам источником содержимого, показано на рис. 12.10. На рис. 12.11 помещено изображение — образец стиля.



**Рис. 12.10.** Изображение с содержимым: Сан-Франциско, район Ноб-Хилл



**Рис. 12.11.** Образец стиля: картина Ван Гога «Звездная ночь»

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.

#### Листинг 12.17. Вспомогательные функции

```
import numpy as np

def preprocess_image(image_path): ←
    img = keras.utils.load_img( ←
        image_path, target_size=(img_height, img_width)) ←
    img = keras.utils.img_to_array(img) ←
    img = np.expand_dims(img, axis=0) ←
    img = keras.applications.vgg19.preprocess_input(img) ←
    return img ←

def deprocess_image(img): ←
    img = img.reshape((img_height, img_width, 3)) ←
    img[:, :, 0] += 103.939 ←
    img[:, :, 1] += 116.779 ←
    img[:, :, 2] += 123.68 ←
    img = img[:, :, ::-1] ←
    img = np.clip(img, 0, 255).astype("uint8") ←
    return img ←
```

Открывает изображение, изменяет его размер и преобразует в соответствующий массив

Вспомогательная функция для преобразования массива NumPy в допустимое изображение

Центрировать относительно нуля путем удаления среднего значения пикселя из ImageNet. Это отменяет преобразование, выполненное vgg19.preprocess\_input

Конвертировать изображения из BGR в RGB. Также отменяет преобразование, выполненное vgg19.preprocess\_input

Настроим сеть VGG19. По аналогии с реализацией DeepDream используем предварительно обученную сверточную сеть и создадим модель, извлекающую признаки и возвращающую активации промежуточных слоев — на этот раз всех слоев.

#### Листинг 12.18. Использование предварительно обученной модели VGG19 для извлечения признаков

```
model = keras.applications.vgg19.VGG19(weights="imagenet", include_top=False) ←

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers]) ←
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict) ←
```

Создать модель VGG19 с загруженными весами, полученными в результате обучения на наборе ImageNet

Модель, возвращающая значения активаций всех целевых слоев (в виде словаря)

Теперь определим функцию потерь содержимого, которая позволит гарантировать сходство представлений целевого и сгенерированного изображений в верхнем слое сети VGG19.

#### Листинг 12.19. Функция потерь содержимого

```
def content_loss(base_img, combination_img):
    return tf.reduce_sum(tf.square(combination_img - base_img))
```

Далее приводится функция потерь стиля. Она использует вспомогательную функцию для вычисления матрицы Грама из входной матрицы: матрицы корреляций, найденных в матрице оригинальных признаков.

#### Листинг 12.20. Функция потерь стиля

```
def gram_matrix(x):
    x = tf.transpose(x, (2, 0, 1))
    features = tf.reshape(x, (tf.shape(x)[0], -1))
    gram = tf.matmul(features, tf.transpose(features))
    return gram

def style_loss(style_img, combination_img):
    S = gram_matrix(style_img)
    C = gram_matrix(combination_img)
    channels = 3
    size = img_height * img_width
    return tf.reduce_sum(tf.square(S - C)) / (4.0 * (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 12.21. Функция общей потери вариации

```
def total_variation_loss(x):
    a = tf.square(
        x[:, :, img_height - 1, : img_width - 1, :] - x[:, 1:, : img_width - 1, :]
    )
    b = tf.square(
        x[:, :, img_height - 1, : img_width - 1, :] - x[:, :, img_height - 1, 1:, :]
    )
    return tf.reduce_sum(tf.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 — список слоев, включающий в себя нижние и верхние слои. Общая потеря вариации добавляется в конец.

В зависимости от используемых изображений с целевым содержимым и образом 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 12.22.** Функция общей потери вариации, которая будет минимизироваться

```

style_layer_names = [ "block1_conv1",
                      "block2_conv1",
                      "block3_conv1",
                      "block4_conv1",
                      "block5_conv1",
]
content_layer_name = "block5_conv2"           ← Слой, используемый для вычисления
total_variation_weight = 1e-6                ← Вес вклада общей потери вариации
style_weight = 1e-6                          ← Вес вклада потери стиля
content_weight = 2.5e-8                     ← Вес вклада потери содержимого

def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0)
    features = feature_extractor(input_tensor)   | Инициализация
    loss = tf.zeros(shape=())                  | потери нулями
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(
        base_image_features, combination_features)
    )                                         | Добавление потери
    for layer_name in style_layer_names:
        layer_features = features[layer_name]
        style_reference_features = layer_features[1, :, :, :]
        combination_features = layer_features[2, :, :, :]
        style_loss_value = style_loss(
            style_reference_features, combination_features)
        loss += (style_weight / len(style_layer_names)) * style_loss_value
        )                                         | Добавление потери стиля
    loss += total_variation_weight * total_variation_loss(combination_image) ←
    return loss                                | Добавление общей
                                                | потери вариации

```

Наконец, настроим процесс градиентного восхождения. В оригинальной статье Гатиса оптимизация проводится с использованием алгоритма L-BFGS, но он недоступен в TensorFlow, поэтому мы выполним обычный мини-пакетный градиентный спуск с оптимизатором SGD. При этом мы будем использовать особенность оптимизатора, которую вы еще не видели: возможность планирования скорости обучения. Мы воспользуемся ею, чтобы постепенно снижать скорость обучения с очень высокого значения (100) до гораздо меньшего конечного значения (около 20). Так мы добьемся быстрого прогресса на ранних этапах обучения, а затем будем действовать более осторожно по мере приближения к минимуму потерь.

**Листинг 12.23.** Настройка процесса градиентного спуска

```

import tensorflow as tf
@tf.function
def compute_loss_and_grads(
    combination_image, base_image, style_reference_image):
    with tf.GradientTape() as tape:
        loss = compute_loss(
            combination_image, base_image, style_reference_image)
        grads = tape.gradient(loss, combination_image)
    return loss, grads

optimizer = keras.optimizers.SGD(
    keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)
base_image = preprocess_image(base_image_path)
style_reference_image = preprocess_image(style_reference_image_path)
combination_image = tf.Variable(preprocess_image(base_image_path)) ←

iterations = 4000
for i in range(1, iterations + 1):
    loss, grads = compute_loss_and_grads(
        combination_image, base_image, style_reference_image
    )
    optimizer.apply_gradients([(grads, combination_image)]) ←
    if i % 100 == 0:
        print(f"Iteration {i}: loss={loss:.2f}")
        img = deprocess_image(combination_image.numpy())
        fname = f"combination_image_at_iteration_{i}.png"
        keras.utils.save_img(fname, img) ←

```

Чтобы ускорить обучение, скомпилируем его как `tf.function`

На первых этапах используем скорость обучения 100, а затем будем уменьшать ее на 4 % через каждые 100 шагов

Используем `Variable` для хранения комбинированного изображения, которое будет изменяться в процессе обучения

Обновим комбинированное изображение в направлении уменьшения потери передачи стиля

Сохраним комбинированное изображение через регулярные интервалы

На рис. 12.12 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 из 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, использовав метод, описанный здесь, а затем обучить простую сверточную сеть данному конкретному преобразованию стиля. После этого можно будет почти мгновенно стилизовать любое изображение: для этого потребуется просто пропустить его через эту маленькую сверточную сеть.



**Рис. 12.12.** Результат передачи стиля

#### 12.3.4. 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.

## 12.4. ГЕНЕРИРОВАНИЕ ИЗОБРАЖЕНИЙ С ВАРИАЦИОННЫМИ АВТОКОДИРОВЩИКАМИ

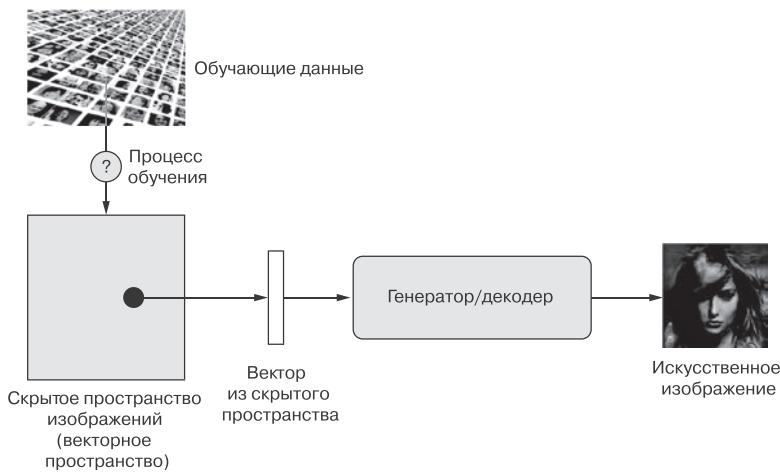
Наиболее популярным и успешным применением художественных возможностей ИИ в настоящее время является генерирование изображений: изучение скрытых визуальных пространств и выбор шаблонов из них для создания совершенно новых изображений путем интерполяции существующих — изображений воображаемых людей, мест, кошек и собак и т. д.

В этом и в следующем разделах мы рассмотрим некоторые высокоуровневые 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.

### 12.4.1. 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 пикселей), 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. 12.13, 12.14). Эти новые изображения являются промежуточными интерполяциями между обучающими изображениями.

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.



**Рис. 12.13.** Обучение скрытого векторного пространства изображений и его использование для создания новых изображений



**Рис. 12.14.** Непрерывное пространство лиц, сгенерированное Томом Уайтом с использованием VAE

## 12.4.2. Концептуальные векторы для редактирования изображений

Мы уже обсуждали идею *концептуальных векторов*, когда рассматривали векторные представления слов в главе 11. 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. 12.15 приводится пример вектора «улыбка» — концептуального вектора, обнаруженного Томом Уайтом 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. 12.15.** Вектор «улыбка»

### 12.4.3. Вариационные автокодировщики

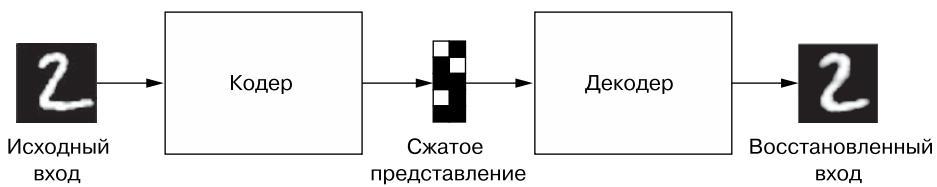
Вариационные автокодировщики, открытые одновременно Дидериком Кингмой и Максом Веллингом в декабре 2013 года<sup>1</sup> и Данило Йименезом Резенде, Шакиром Мохамедом и Дааном Вирстрой в январе 2014 года<sup>2</sup>, являются разновидностью генеративных моделей, которая особенно хорошо подходит для редактирования изображений посредством концептуальных векторов. Они предлагают современный подход к автокодировщикам — разновидности сетей, целью которых является кодирование входного малоразмерного скрытого про-

<sup>1</sup> Kingma D. P., Welling M. Auto-Encoding Variational Bayes // arXiv, 2013, <https://arxiv.org/abs/1312.6114>.

<sup>2</sup> Rezende D.J., Shakir M., Daan W. Stochastic Backpropagation and Approximate Inference in Deep Generative Models // arXiv, 2014, <https://arxiv.org/abs/1401.4082>.

странства и последующего его декодирования, — сочетающим 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. 12.16). 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. 12.16.** 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 процесс декодирования толику статистического волшебства, извлекая 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 оригинальный вход (рис. 12.17). Стохастичность этого процесса повышает надежность и заставляет скрытое пространство кодировать значимые представления: каждая точка, выбранная в скрытом пространстве, декодируется в допустимый вывод.

Вот как работает вариационный автокодировщик с технической точки зрения.

1. Модуль кодирования превращает входной образец изображения `input_img` в два параметра в скрытом пространстве, `z_mean` и `z_log_variance`.
2. Вы выбираете из скрытого нормального распределения произвольную точку `z` для генерации входного изображения как  $z = z\_mean + \exp(z\_log\_variance) * \epsilon$ , где `epsilon` — это случайный тензор небольших значений.
3. Модуль декодера отображает эту точку из скрытого пространства обратно в оригинальное изображение.

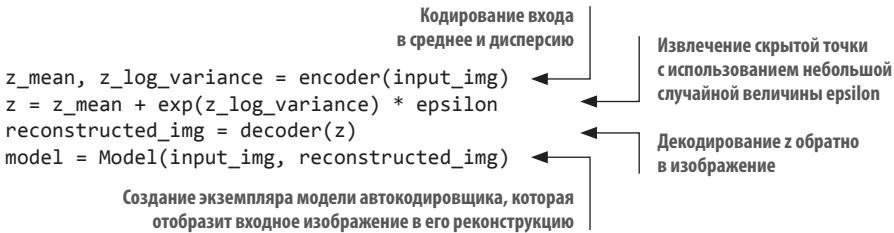


**Рис. 12.17.** Вариационный автокодировщик отображает изображение в два вектора, `z_mean` и `z_log_sigma`, которые определяют распределение вероятности в скрытом пространстве, используемое для выбора точки для декодирования

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: на *потерях восстановления* (reconstruction loss), которые заставляют декодированные образцы совпадать с исходными входами, и *потерях регуляризации* (regularization loss), которые помогают извлекать хорошо сформированные скрытые пространства и ослабляют проблему переобучения на обучающих

данных. Давайте пройдемся по реализации вариационного автокодировщика в Keras. Схематически она выглядит так:



Затем можно обучить модель, используя потери восстановления и потери регуляризации. Для вычисления потери регуляризации обычно применяется выражение (расхождение Кульбака — Лейблера), предназначенное для подталкивания распределения выходных данных кодировщика к нормальному распределению с центром в точке 0. Это дает кодировщику разумное предположение о структуре скрытого пространства, которое он моделирует.

А теперь давайте посмотрим, как выглядит реализация вариационного автокодировщика (VAE) на практике!

#### 12.4.4. Реализация VAE в Keras

Далее мы реализуем вариационный автокодировщик, способный генерировать изображения цифр, похожие на изображения в наборе данных MNIST. Он будет состоять из трех частей:

- сети кодировщика, превращающей реальное изображение в среднее и дисперсию в скрытом пространстве;
- слоя выбора образца, принимающего среднее значение и дисперсию и использующего их для выбора случайной точки в скрытом пространстве;
- сети декодера, превращающей точку из скрытого пространства обратно в изображения.

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  $z\_log\_var$ . Важно отметить, что для уменьшения разрешения карт признаков вместо выбора максимального из соседних значений мы будем выполнять выборку с определенным шагом. В последний раз мы делали это в примере сегментации изображения в главе 9. Напомню, что в общем случае выборка с определенным шагом предпочтительнее выбора максимального

из соседних значений, когда важно сохранить *информацию о местоположении*, то есть о том, где что-то находится в изображении. Это как раз наш случай, ведь модель должна закодировать изображение так, чтобы потом кодировку можно было использовать для восстановления действительного изображения.

#### Листинг 12.24. Сеть кодировщика VAE

```
from tensorflow import keras
from tensorflow.keras import layers
latent_dim = 2
encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Размерность скрытого пространства:  
двумерная плоскость

Входное изображение  
кодируется в эти  
два параметра

Вот сводная информация о получившейся модели:

```
>>> encoder.summary()
Model: "encoder"
```

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	[None, 28, 28, 1]	0	
conv2d (Conv2D)	(None, 14, 14, 32)	320	input_1[0][0]
conv2d_1 (Conv2D)	(None, 7, 7, 64)	18496	conv2d[0][0]
flatten (Flatten)	(None, 3136)	0	conv2d_1[0][0]
dense (Dense)	(None, 16)	50192	flatten[0][0]
z_mean (Dense)	(None, 2)	34	dense[0][0]
z_log_var (Dense)	(None, 2)	34	dense[0][0]
<hr/>			
Total params: 69,076			
Trainable params: 69,076			
Non-trainable params: 0			

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.

**Листинг 12.25.** Слой выбора точки из скрытого пространства

```
import tensorflow as tf

class Sampler(layers.Layer):
    def call(self, z_mean, z_log_var):
        batch_size = tf.shape(z_mean)[0]
        z_size = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch_size, z_size)) ← Выбрать набор
                                                               случайных векторов
                                                               из нормального
                                                               распределения
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon ← Применить
                                                       формулу
                                                       выборки VAE
```

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 12.26.** Сеть декодера VAE, отображающая точки из скрытого пространства в изображения

```
latent_inputs = keras.Input(shape=(latent_dim,)) ← Произвести столько же коэффициентов, сколько
                                                 имеется на уровне слоя Flatten в кодировщике
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs) ← Передача z на вход
x = layers.Reshape((7, 7, 64))(x) ← Восстановить слой Flatten кодировщика
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x) ← Восстановить слой Conv2D
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder") ← кодировщика
                                                                     Выход будет иметь форму (28, 28, 1)
```

Вот сводная информация о получившейся модели:

```
>>> decoder.summary()
Model: "decoder"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[(None, 2)]	0
dense_1 (Dense)	(None, 3136)	9408
reshape (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose (Conv2DTran	(None, 14, 14, 64)	36928
conv2d_transpose_1 (Conv2DTr	(None, 28, 28, 32)	18464
conv2d_2 (Conv2D)	(None, 28, 28, 1)	289
<hr/>		
Total params: 65,089		
Trainable params: 65,089		
Non-trainable params: 0		

Теперь реализуем саму модель VAE. Это наш первый пример модели с обучением без учителя (в отличие от автокодировщика — примера модели с *самообучением*, поскольку в качестве целей он использует свои входные данные). Всякий раз при отступлении от классического способа обучения с учителем обычно создается подкласс класса `Model` и реализуется свой метод `train_step()`, уточняющий новую логику обучения, — рабочий процесс, с которым вы познакомились в главе 7. Этим мы и займемся.

### Листинг 12.27. Модель VAE с нестандартным методом `train_step()`

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampler = Sampler()
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss")
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [self.total_loss_tracker,
                self.reconstruction_loss_tracker,
                self.kl_loss_tracker]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var = self.encoder(data)
            z = self.sampler(z_mean, z_log_var)
            reconstruction = decoder(z)
            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction),
                    axis=(1, 2))
            )
            k1_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
                              tf.exp(z_log_var))
            total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
            grads = tape.gradient(total_loss, self.trainable_weights)
            self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
            self.total_loss_tracker.update_state(total_loss)
            self.reconstruction_loss_tracker.update_state(reconstruction_loss)
            self.kl_loss_tracker.update_state(kl_loss)
        return {
            "total_loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

Эти метрики используются для слежения за средними значениями потерь в каждой эпохе

Перечисляем метрики в свойстве `metrics`, чтобы модель могла сбрасывать их после каждой эпохи (или между вызовами `fit()/evaluate()`)

Суммируем потери при реконструкции по пространственным измерениям (оси 1 и 2) и берем их средние значения

Добавляем член регуляризации (расхождение Кульбака — Лейблера)

Наконец, мы готовы создать и обучить экземпляр модели на изображениях цифр из набора MNIST. 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 12.28. Обучение VAE

```
import numpy as np

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(), run_eagerly=True)
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

Обучение выполняется на полном наборе данных MNIST, поэтому мы объединили обучающий и контрольный наборы

Помните, что мы не передаем цели в метод `fit()`, поскольку они не используются в `train_step()`

Обратите внимание, что мы не передаем аргумент `loss` в вызов `compile()`, потому что вычисление потерь выполняется в методе `train_step()`

После обучения такой модели мы можем использовать сеть `decoder` для превращения произвольных векторов из скрытого пространства в изображения.

### Листинг 12.29. Выбор сетки с изображениями из двумерного скрытого пространства

```
import matplotlib.pyplot as plt

n = 30          ← Отобразить сетку 30 × 30 цифр
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-1, 1, n)           ← Выбрать ячейки, линейно
grid_y = np.linspace(-1, 1, n)[::-1]      ← распределенные в двумерной сетке

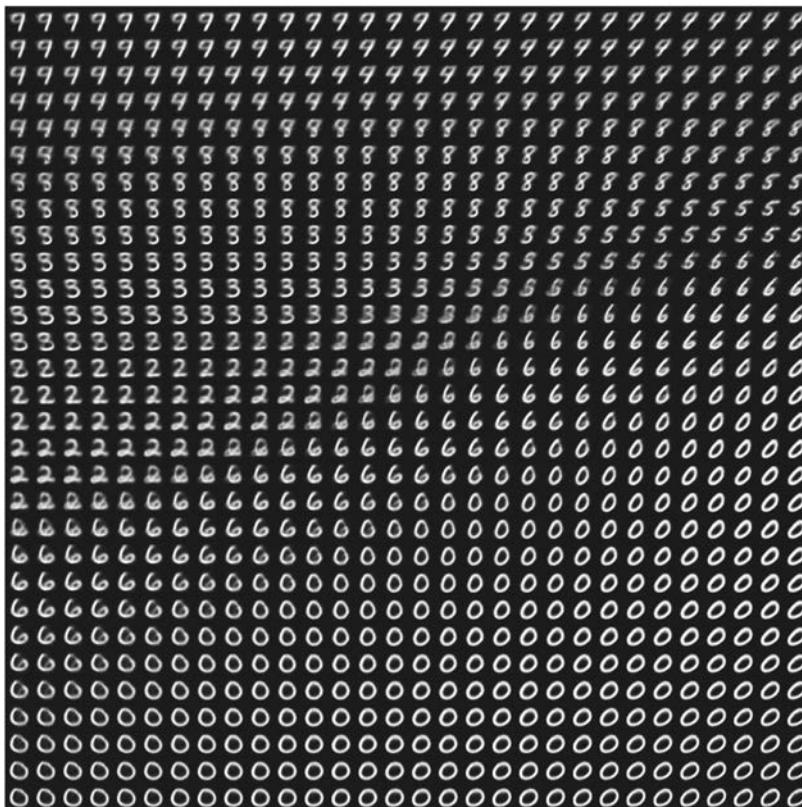
for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])      ← Выполнить обход
        x_decoded = vae.decoder.predict(z_sample)  ← ячеек в сетке
        digit = x_decoded[0].reshape(digit_size, digit_size)  ← Для каждой ячейки
        figure[                           ← выбрать цифру и добавить
            i * digit_size : (i + 1) * digit_size,
            j * digit_size : (j + 1) * digit_size,
        ] = digit
```

```

plt.figure(figsize=(15, 15))
start_range = digit_size // 2
end_range = n * digit_size + start_range
pixel_range = np.arange(start_range, end_range, digit_size)
sample_range_x = np.round(grid_x, 1)
sample_range_y = np.round(grid_y, 1)
plt.xticks(pixel_range, sample_range_x)
plt.yticks(pixel_range, sample_range_y)
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.axis("off")
plt.imshow(figure, cmap="Greys_r")

```

Сетка выбранных цифр (рис. 12.18) 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 «пятерочности», «единичности» и т. д.



**Рис. 12.18.** Сетка с цифрами, декодированными из скрытого пространства

В следующем разделе мы подробно рассмотрим еще один важный инструмент создания искусственных изображений: генеративно-состязательные сети (Generative Adversarial Networks, GAN).

#### 12.4.5. 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.

## 12.5. ВВЕДЕНИЕ В ГЕНЕРАТИВНО-СОСТАЗАТЕЛЬНЫЕ СЕТИ

Генеративно-состязательные сети (Generative Adversarial Networks, GAN), впервые представленные в 2014 году Яном Гудфеллоу и его коллегами<sup>1</sup>, — альтернатива вариационным автокодировщикам для выделения скрытых пространств изображений. Они позволяют генерировать очень реалистичные искусственные изображения, статистически неотличимые от настоящих.

<sup>1</sup> Goodfellow I. et al. Generative Adversarial Networks // arXiv, 2014, <https://arxiv.org/abs/1406.2661>.

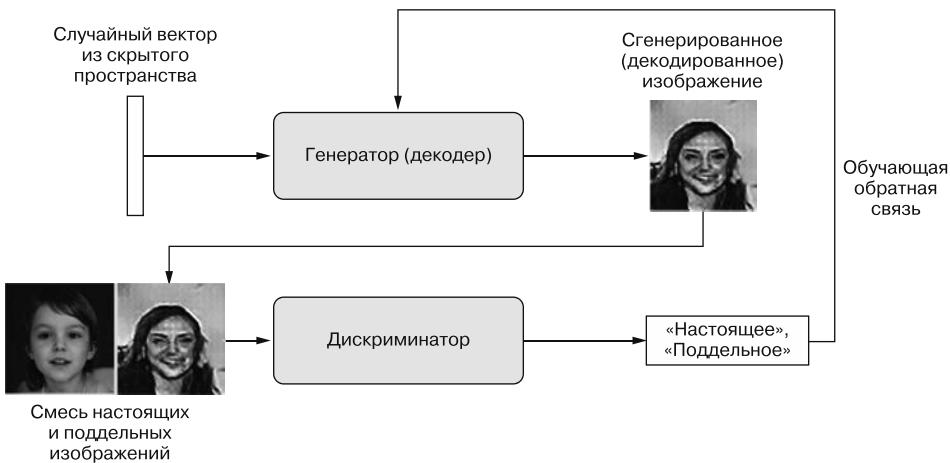
Чтобы проще было понять суть генеративно-состязательной сети, вообразите фальсификатора, пытающегося подделать картину Пикассо. Сначала он довольно плохо справляется с задачей. Он показывает свои подделки вместе с подлинниками Пикассо продавцу произведений искусства. Продавец оценивает подлинность картин 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 обчается обманывать сеть-дискриминатор и, соответственно, учится создавать все более реалистичные изображения: поддельные изображения, почти неотличимые от настоящих (рис. 12.19). 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 генеративно-состязательную сеть, требуется приложить большие усилия по настройке архитектуры модели и параметров обучения.



**Рис. 12.19.** Генератор преобразует случайные скрытые векторы в изображения, а дискриминатор стремится отличить настоящие изображения от сгенерированных искусственно. Генератор обучается обманывать дискриминатор

### 12.5.1. Реализация простейшей генеративно-состязательной сети

Далее я расскажу, как реализовать простейшую генеративно-состязательную сеть с использованием Keras. Стоит отметить, что сети этого вида очень сложны, и подробное описание технических деталей архитектур, подобных архитектуре StyleGAN2, сгенерировавшей изображения на рис. 12.20, выходит далеко за рамки этой книги. Данная простейшая реализация — это *глубокая сверточная генеративно-состязательная сеть* (Deep Convolutional GAN, DCGAN), в которой генератор и дискриминатор являются глубокими сверточными сетями.



**Рис. 12.20.** Скрытое пространство жителей. Изображения предоставлены автором сайта <https://thispersondoesnotexist.com> Филиппом Ваном. Для их создания использовалась модель StyleGAN2, разработанная Каррасом с коллегами, <https://arxiv.org/abs/1912.04958>

Мы будем обучать GAN на изображениях из набора Large-scale CelebFaces Attributes (известного как CelebA), содержащего 200 000 изображений знаменитостей (<http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>). Чтобы ускорить обучение, мы уменьшим размеры изображений до  $64 \times 64$ , то есть обучим сеть генерировать изображения человеческих лиц размером  $64 \times 64$ .

В общих чертах GAN выглядит примерно так:

- сеть `generator` отображает векторы с формой (размерность\_скрытого\_пространства,) в изображения с формой (64, 64, 3);
- сеть `discriminator` отображает изображения с формой (64, 64, 3) в оценку вероятности, что изображение является настоящим;
- сеть `gan` объединяет генератор и дискриминатор `gan(x) = discriminator(generator(x))`, иными словами, отображает скрытое пространство векторов в оценку реализма этих скрытых векторов, декодированных генератором;
- мы обучим дискриминатор на примерах реальных и искусственных изображений, отмеченных метками «настоящее»/«поддельное», как самую обычную модель классификации изображений;
- для обучения генератора 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.

### 12.5.2. Набор хитростей

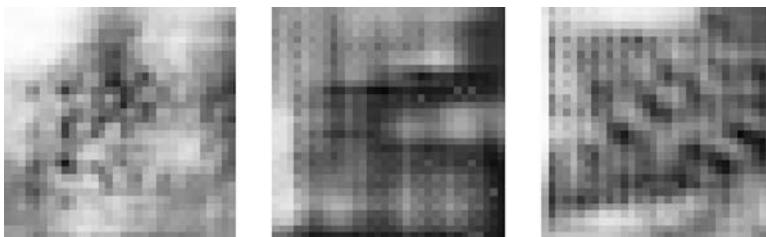
Процесс обучения и настройки генеративно-состязательных сетей очень сложен. Однако есть несколько хитростей, которые следует знать и помнить. Как и многое другое в глубоком обучении, это больше алхимия, чем наука: все хитрости, описываемые далее, выявлены экспериментальным путем и не имеют теоретического 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 дискриминаторе вместо объединения мы будем уменьшать разрешение карт признаков за счет изменения шага выборки, по аналогии с вариационным автокодировщиком (VAE).
- 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 будем вводить случайный компонент, 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 я рекомендую увеличивать шаг свертки, а вместо функции активации `relu` использовать 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 пикселей в генераторе (рис. 12.21). Для их устранения мы будем выбирать размер ядра, кратный размеру шага, при каждом использовании разреженных слоев Conv2DTranspose или Conv2D в генераторе и дискrimинаторе.



**Рис. 12.21.** Артефакты типа «шахматная доска», вызванные несовпадением размеров шага и ядра, из-за чего происходит неравномерный охват пространства пикселей: одна из многих тонкостей GAN, доставляющих хлопоты

### 12.5.3. Получение набора данных CelebA

Загрузить набор данных можно вручную с веб-сайта <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>. Если вы используете Colab, выполните следующие команды — они загрузят данные с Google Drive и распакуют их.

**Листинг 12.30.** Получение набора данных CelebA

```
!mkdir celeba_gan ← Создать рабочий каталог  
!gdown --id 107m1010EJjLE5QxLZiM9Fpj570j6e684 -O celeba_gan/data.zip ← Загрузить архив с данными с помощью утилиты gdown  
!unzip -qq celeba_gan/data.zip -d celeba_gan ← Распаковать данные
```

После распаковки изображений в каталог можно превратить каталог `image_dataset_from_directory` в объект набора данных. Так как нам нужны только изображения — без меток, — передадим аргумент `label_mode=None`.

**Листинг 12.31.** Создание объекта набора данных на основе каталога с изображениями

```
from tensorflow import keras
dataset = keras.utils_dataset_from_directory(
    "celeba_gan",
    label_mode=None,           ← Возвращаться должны только
    image_size=(64, 64),       ← изображения, без меток
    batch_size=32,
    smart_resize=True)         ← Уменьшить размеры изображений до  $64 \times 64$ , используя
                                комбинацию операций обрезки и масштабирования для сохранения
                                пропорций. Нам не нужно, чтобы пропорции лиц искажались!
```

Наконец, масштабируем значения, представляющие пиксели, в диапазон [0–1].

**Листинг 12.32.** Масштабирование значений пикселей в изображениях

```
dataset = dataset.map(lambda x: x / 255.)
```

Для вывода выбранного изображения можно использовать следующий код.

**Листинг 12.33.** Вывод на экран первого изображения

```
import matplotlib.pyplot as plt
for x in dataset:
    plt.axis("off")
    plt.imshow((x.numpy() * 255).astype("int32")[0])
    break
```

## 12.5.4. Дискриминатор

Теперь перейдем к модели дискриминатора, которая принимает на входе изображение-кандидат (реальное или искусственное) и относит его к одному из двух классов: «подделка» или «настоящее, имеющееся в обучающем наборе». Одна из многих проблем, часто возникающих в сетях GAN, — генератор создает изображения, которые выглядят как шум. Одно из возможных решений — использовать прореживание в дискриминаторе, что мы и сделаем здесь.

**Листинг 12.34.** Сеть дискриминатора в GAN

```
from tensorflow.keras import layers
discriminator = keras.Sequential([
[
```

```

    keras.Input(shape=(64, 64, 3)),
    layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Flatten(),
    layers.Dropout(0.2),           ← Слой прореживания:
    layers.Dense(1, activation="sigmoid"),
],
name="discriminator",
)

```

Вот сводная информация о получившейся модели дискриминатора:

```
>>> discriminator.summary()
Model: "discriminator"
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 32, 32, 64)	3136
leaky_re_lu (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 16, 16, 128)	131200
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	262272
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 128)	0
flatten (Flatten)	(None, 8192)	0
dropout (Dropout)	(None, 8192)	0
dense (Dense)	(None, 1)	8193
<hr/>		
Total params: 404,801		
Trainable params: 404,801		
Non-trainable params: 0		

## 12.5.5. Генератор

Затем реализуем модель `generator`, преобразующую вектор (из скрытого пространства — во время обучения он будет выбираться случайно) в изображение-кандидат.

**Листинг 12.35.** Сеть генератора в GAN

```

latent_dim = 128           ← Скрытое пространство будет
                            состоять из 128-мерных
                            векторов

generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(8 * 8 * 128),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(3, kernel_size=5, padding="same", activation="sigmoid"),
    ],
    name="generator",
)

```

Произвести такое же количество коэффициентов, как в слое Flatten в кодировщике

Использовать слой LeakyReLU в качестве функции активации

Восстановить слой Flatten кодировщика

Выход будет иметь форму (28, 28, 1)

Восстановить слой Conv2D кодировщика

Вот сводная информация о получившейся модели генератора:

```
>>> generator.summary()
Model: "generator"
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_1 (Dense)	(None, 8192)	1056768
reshape (Reshape)	(None, 8, 8, 128)	0
conv2d_transpose (Conv2DTran	(None, 16, 16, 128)	262272
leaky_re_lu_3 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_transpose_1 (Conv2DTr	(None, 32, 32, 256)	524544
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 256)	0
conv2d_transpose_2 (Conv2DTr	(None, 64, 64, 512)	2097664
leaky_re_lu_5 (LeakyReLU)	(None, 64, 64, 512)	0
conv2d_3 (Conv2D)	(None, 64, 64, 3)	38403
<hr/>		
Total params: 3,979,651		
Trainable params: 3,979,651		
Non-trainable params: 0		

## 12.5.6. Состязательная сеть

Наконец, перейдем к состязательной сети, объединяющей генератор и дискриминатор. В процессе обучения эта модель будет смещать 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 смещать веса в модели `generator` так, чтобы увеличить вероятность получить от дискриминатора ответ «настоящее», когда тот будет просматривать поддельное изображение.

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». Данное действие 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 схему. Как и в примере с VAE, определим подкласс класса `Model` с нестандартным методом `train_step()`. Обратите внимание, что на этот раз мы будем использовать два оптимизатора (один для генератора и один для дискриминатора), поэтому мы также переопределим метод `compile()`, чтобы реализовать возможность передачи двух оптимизаторов.

**Листинг 12.36.** Подкласс `Gan` класса `Model`

```
import tensorflow as tf

class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
```

```

self.d_loss_metric = keras.metrics.Mean(name="d_loss")
self.g_loss_metric = keras.metrics.Mean(name="g_loss") |



def compile(self, d_optimizer, g_optimizer, loss_fn):
    super(GAN, self).compile()
    self.d_optimizer = d_optimizer
    self.g_optimizer = g_optimizer
    self.loss_fn = loss_fn
    Настройка метрик  
для отслеживания  
двух потерь  
в каждой эпохе  
обучения |



@property
def metrics(self):
    return [self.d_loss_metric, self.g_loss_metric] |



def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    random_latent_vectors = tf.random.normal( | Выбор случайных
                                                | точек из скрытого
                                                | пространства
    shape=(batch_size, self.latent_dim)) |
    generated_images = self.generator(random_latent_vectors)
    combined_images = tf.concat([generated_images, real_images], axis=0) |
    labels = tf.concat( | Добавление случайного шума
        [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], | в метки — это важный шаг!
        axis=0
    )
    labels += 0.05 * tf.random.uniform(tf.shape(labels)) |



with tf.GradientTape() as tape:
    predictions = self.discriminator(combined_images)
    d_loss = self.loss_fn(labels, predictions)
grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
self.d_optimizer.apply_gradients(
    zip(grads, self.discriminator.trainable_weights))
    Сбор меток,  
отличающих  
настоящие  
изображения  
от поддельных |



random_latent_vectors = tf.random.normal( | Выбор случайных точек
                                            | из скрытого пространства
    shape=(batch_size, self.latent_dim)) |



misleading_labels = tf.zeros((batch_size, 1)) | Сбор меток, говорящих: «Все эти изображения настоящие» (это ложь!)



with tf.GradientTape() as tape:
    predictions = self.discriminator(
        self.generator(random_latent_vectors))
    g_loss = self.loss_fn(misleading_labels, predictions)
grads = tape.gradient(g_loss, self.generator.trainable_weights)
self.g_optimizer.apply_gradients(
    zip(grads, self.generator.trainable_weights)) |



self.d_loss_metric.update_state(d_loss)
self.g_loss_metric.update_state(g_loss)
return {"d_loss": self.d_loss_metric.result(),
        "g_loss": self.g_loss_metric.result()} |



Объединение  
поддельных  
изображений  
с настоящими |



Декодирование  
точек в поддельные  
изображения |



Сбор меток,  
отличающих  
настоящие  
изображения  
от поддельных |



Обучение  
дискриминатора |



Выбор случайных точек  
из скрытого пространства |



Обучение  
генератора |

```

Перед началом обучения настроим обратный вызов для отслеживания результатов: он будет с помощью генератора создавать и сохранять несколько поддельных изображений в конце каждой эпохи.

**Листинг 12.37.** Обратный вызов, генерирующий образцы изображений в процессе обучения

```
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=3, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.normal(
            shape=(self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
        for i in range(self.num_img):
            img = keras.utils.array_to_img(generated_images[i])
            img.save(f"generated_img_{epoch:03d}_{i}.png")
```

Наконец, запустим обучение.

**Листинг 12.38.** Компиляция и обучение сети GAN

```
epochs = 100 ← | Интересные результаты начнут
                  | появляться примерно после 20-й эпохи

gan = GAN(discriminator=discriminator, generator=generator,
           latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(),
)
gan.fit(
    dataset, epochs=epochs,
    callbacks=[GANMonitor(num_img=10, latent_dim=latent_dim)]
)
```

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.

На рис. 12.22 показаны примеры изображений, которые наша сеть GAN смогла сгенерировать после 30 эпох обучения.



**Рис. 12.22.** Несколько изображений, сгенерированных после 30 эпох обучения

### 12.5.7. 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 случаев практического применения, таких как редактирование изображений с использованием концептуальных векторов.

Эти несколько методов охватывают лишь самые основы данного быстроразвивающегося направления. Вам еще многое предстоит узнать — генеративное глубокое обучение достойно отдельной книги.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Модели типа «последовательность в последовательность» позволяют генерировать последовательности данных по одному элементу за раз. Их можно использовать для создания текстов, а также мелодий — нота за нотой — и любых других подобных временных последовательностей.
- Алгоритм DeepDream максимизирует активации слоя сверточной сети через градиентное восхождение во входном пространстве.
- Алгоритм передачи стиля объединяет изображение с целевым контентом и образец стиля методом градиентного спуска, чтобы создать изображение с высокоуровневыми характеристиками изображения с целевым контентом и локальными характеристиками образца стиля.
- Генеративно-состязательные сети и вариационные автокодировщики исследуют скрытое пространство изображений и затем могут генерировать совершенно новые образы, извлекая образцы из скрытого пространства. *Концептуальные векторы* в скрытом пространстве можно использовать даже для редактирования изображений.

# 13

## *Методы и приемы для применения на практике*

---

### **В этой главе**

- ✓ Настройка гиперпараметров.
- ✓ Ансамблирование моделей.
- ✓ Обучение со смешанной точностью.
- ✓ Обучение моделей Keras на нескольких GPU или TPU.

Итак, к настоящему моменту вы продвинулись достаточно далеко. Вы уже умеете обучать модели классификации или сегментации изображений, классификации или регрессии векторных данных, прогнозированию временных рядов, классификации текста, преобразованию одних последовательностей в другие последовательности и даже генеративные модели для создания текста и изображений. Вы изучили практически все основы.

Однако до сих пор ваши модели обучались на небольших наборах данных, с одним графическим процессором — и, как правило, не достигали максимально возможного качества на всех рассматривавшихся наборах данных. Впрочем, такой результат вполне ожидаем — книга перед вами является всего лишь вводной. Если вы собираетесь выйти в реальный мир и добиться чего-то значительного в совершенно новых задачах, вам все равно придется преодолеть небольшой разрыв.

Преодолению этого разрыва и посвящена предпоследняя глава данной книги. В ней я постараюсь передать толику передового опыта, который понадобится вам на пути от студента, изучающего машинное обучение, до полноценного инженера по машинному обучению. Мы рассмотрим основные методы систематического улучшения качества моделей: настройку гиперпараметров и ансамблирование моделей. Затем поговорим о том, как расширить масштабы и ускорить обучение моделей с применением методов обучения со смешанной точностью и с использованием нескольких GPU и TPU, а также удаленных вычислительных ресурсов в облаке.

## 13.1. ПОЛУЧЕНИЕ МАКСИМАЛЬНОЙ ОТДАЧИ ОТ МОДЕЛЕЙ

Слепая аprobация различных конфигураций позволяет добиться желаемого, если все, что вам нужно, — чтобы модель «работала нормально». В этом разделе мы перейдем от «работает нормально» к «работает отлично и выигрывает соревнования по машинному обучению», рассмотрев набор методов, применяемых при создании современных моделей глубокого обучения.

### 13.1.1. 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 гиперпараметров 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.

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.

- 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 градиентный спуск.
- Вычисление сигнала обратной связи (действительно ли данный набор гиперпараметров ведет к улучшению качества модели для данной задачи?) может обходиться очень дорого: для этого нужно создать и обучить новую модель с нуля.
- Сигнал обратной связи может искажаться посторонними шумами: если модель, обученная с новым набором гиперпараметров, оказалась на 0,2 % лучше прежней, как определить, чем обусловлен этот прирост — лучшей конфигурацией модели или простым везением в выборе начальных значений весов?

К счастью, существует инструмент, упрощающий настройку гиперпараметров: KerasTuner. Давайте познакомимся с ним.

## KerasTuner

Начнем с установки KerasTuner:

```
!pip install keras-tuner -q
```

KerasTuner позволяет заменять жестко запрограммированные значения гиперпараметров (такие как `units=32`) диапазоном возможных вариантов, например: `Int(name="units", min_value=16, max_value=64, step=16)`. Это множество вариантов называется *пространством поиска* процесса настройки гиперпараметров.

Чтобы задать пространство поиска, нужно определить функцию создания модели (как показано в следующем листинге). Она принимает параметр `hp`, с помощью которого можно задавать диапазоны гиперпараметров, и возвращает скомпилированную модель Keras.

### Листинг 13.1. Функция создания модели для KerasTuner

```
from tensorflow import keras
from tensorflow.keras import layers

def build_model(hp):
    units = hp.Int(name="units", min_value=16, max_value=64, step=16) ← Выбор гиперпараметров из объекта hp. После
    model = keras.Sequential([                                         выбора эти значения (такие как переменная "units"
        layers.Dense(units, activation="relu"),
        layers.Dense(10, activation="softmax")                         здесь) становятся обычными константами Python
    ])                                                               ← Доступны разные типы
    optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"]) ← гиперпараметров: Int,
    model.compile(                                                 ← Float, Boolean, Choice
        optimizer=optimizer,
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"])
    return model                                                    ← Функция возвращает
                                                                    скомпилированную модель
```

При желании можно использовать более модульный и настраиваемый подход к созданию модели, а также создать подкласс класса `HyperModel` и определить метод `build`, как показано ниже.

### Листинг 13.2. Создание производного класса от HyperModel в KerasTuner

```
import kerastuner as kt

class SimpleMLP(kt.HyperModel):
    def __init__(self, num_classes):
        self.num_classes = num_classes

    def build(self, hp): ← Используя объектно-ориентированный подход, можно
                        настраивать константы модели с помощью аргументов
                        конструктора (вместо жесткого их определения
                        в функции создания модели)
        units = hp.Int(name="units", min_value=16, max_value=64, step=16) ←
        model = keras.Sequential([
            layers.Dense(units, activation="relu"),
            layers.Dense(self.num_classes, activation="softmax")
        ])                                                               ← Метод build() играет ту же роль, что
                                                                    и использовавшаяся прежде функция build_model()
```

```

optimizer = hp.Choice(name="optimizer", values=["rmsprop", "adam"])
model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
return model

hypermodel = SimpleMLP(num_classes=10)

```

Следующим шагом является определение тюнера. Схематически тюнер можно представить как цикл `for`, который многократно:

- выбирает набор значений гиперпараметров;
- вызывает функцию создания модели с этими значениями;
- обучает модель и сохраняет полученные метрики.

KerasTuner имеет несколько встроенных тюнеров — `RandomSearch`, `BayesianOptimization` и `Hyperband`. Попробуем `BayesianOptimization` — тюнер, пытающийся интеллектуально подбирать новые значения гиперпараметров с учетом предыдущего выбора, которые, вероятно, будут обеспечивать более высокое качество модели:

```

tuner = kt.BayesianOptimization(
    build_model,
    objective="val_accuracy",
    max_trials=100,
    executions_per_trial=2,
    directory="mnist_kt_test",
    overwrite=True,
)

```

**Указать функцию создания модели (или экземпляра HyperModel)**

**Указать метрику, которую должен оптимизировать тюнер. Всегда указывайте метрики, получаемые на проверочных данных, потому что цель процесса поиска — найти модель с хорошей общностью!**

**Максимальное количество конфигураций моделей («попыток») для аprobации перед завершением процесса поиска**

**Каталог для сохранения журналов с результатами поиска**

**Чтобы уменьшить дисперсию метрик, можно обучить одну и ту же модель несколько раз и усреднить результаты. executions\_per\_trial определяет количество раундов обучения для каждой испытуемой конфигурации модели (попытки)**

**К вопросу, следует ли перезаписать прежние данные в каталоге перед началом нового поиска. Присвойте этому параметру значение True, если изменили функцию создания модели, и значение False, чтобы возобновить ранее начатый поиск с той же функцией создания модели**

Вывести на экран состояние пространства поиска можно с помощью `search_space_summary()`:

```

>>> tuner.search_space_summary()
Search space summary
Default search space size: 2
units (Int)
{"default": None,
 "conditions": [],
 "min_value": 128,
 "max_value": 1024,
 "step": 128,
 "sampling": None}

```

```
optimizer (Choice)
{"default": "rmsprop",
 "conditions": [],
 "values": ["rmsprop", "adam"],
 "ordered": False}
```

### ЦЕЛЬ МАКСИМИЗАЦИИ И МИНИМИЗАЦИИ

Для встроенных метрик (таких как точность в нашем случае) *направление* (*direction*) изменения метрики (точность должна быть максимальной, а потери — минимальными) KerasTuner определяет автоматически. Однако при использовании своих метрик вы должны указать его явно, например:

```
objective = kt.Objective(
    name="val_accuracy",
    direction="max")
tuner = kt.BayesianOptimization(
    build_model,
    objective=objective,
    ...
)
```

Подготовив все необходимое, можно запустить поиск. Не забудьте передать проверочные данные и убедитесь, что контрольный набор не используется для проверки — иначе модель будет обучаться также на контрольных данных и вы не сможете больше доверять контрольным метрикам:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape((-1, 28 * 28)).astype("float32") / 255
x_test = x_test.reshape((-1, 28 * 28)).astype("float32") / 255
x_train_full = x_train[:] | Зарезервировать для дальнейшего
y_train_full = y_train[:] | использования
num_val_samples = 10000
x_train, x_val = x_train[:-num_val_samples], x_train[-num_val_samples:] |
y_train, y_val = y_train[:-num_val_samples], y_train[-num_val_samples:] |
callbacks = [
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=5),
]
tuner.search( | Данный метод принимает те же
    x_train, y_train, | аргументы, что и fit() (он просто
    batch_size=128, | передает их в вызов функции fit()
    epochs=100, | для каждой новой модели)
    validation_data=(x_val, y_val),
    callbacks=callbacks,
    verbose=2,
)
```

Использовать большое количество эпох (наперед неизвестно, сколько эпох потребуется для каждой модели) и обратный вызов *EarlyStopping*, чтобы остановить обучение, когда возникнет эффект переобучения

Предыдущий пример выполнится всего за несколько минут, так как мы рассматриваем лишь несколько возможных вариантов, а обучение производится на наборе данных MNIST. Однако в реальности с типичным пространством поиска и набором данных для поиска гиперпараметров часто может потребоваться вся ночь или даже несколько дней. Если в процессе поиска произошел сбой, вы всегда сможете перезапустить его — просто укажите в настройках тюнера `overwrite=False`, чтобы он смог возобновить перебор настроек, опираясь на сохраненные на диске журналы.

После завершения поиска можно запросить лучшие конфигурации гиперпараметров и использовать их для создания высококачественных моделей, которые затем можно повторно обучить.

### Листинг 13.3. Запрос лучших конфигураций гиперпараметров

```
top_n = 4
best_hps = tuner.get_best_hyperparameters(top_n) ←
    Вернет список объектов HyperParameter, которые
    можно передать в вызов функции создания модели
```

Часто при повторном обучении моделей желательно включить проверочные данные в обучающий набор, если в дальнейшем не предполагается вносить какие-либо изменения в гиперпараметры: ведь тогда качество модели на этих данных больше не будет оцениваться. В нашем примере мы обучим окончательные модели на всей совокупности обучающих данных из набора MNIST, не резервируя проверочного набора.

Прежде чем приступить к тренировке на полном наборе обучающих данных, нужно установить последний параметр: оптимальное количество эпох обучения. Обычно желательно обучать новые модели дольше, чем во время поиска: использование агрессивного значения `patience` в обратном вызове `EarlyStopping` экономит время в течение поиска, но может привести к малопригодным моделям. Просто используйте проверочный набор для поиска лучшей эпохи:

```
def get_best_epoch(hp):
    model = build_model(hp)
    callbacks=[keras.callbacks.EarlyStopping(
        monitor="val_loss", mode="min", patience=10) ←
    ]
    history = model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=100,
        batch_size=128,
        callbacks=callbacks)
    val_loss_per_epoch = history.history["val_loss"]
    best_epoch = val_loss_per_epoch.index(min(val_loss_per_epoch)) + 1
    print(f"Best epoch: {best_epoch}")
    return best_epoch
```

Обратите внимание  
на необычно высокое  
значение параметра patience

Наконец, проведем обучение на полном наборе данных, задав чуть большее количество эпох, потому что теперь обучение производится на большем количестве данных. В нашем случае увеличим число эпох на 20 %:

```
def get_best_trained_model(hp):
    best_epoch = get_best_epoch(hp)
    model.fit(
        x_train_full, y_train_full,
        batch_size=128, epochs=int(best_epoch * 1.2))
    return model

best_models = []
for hp in best_hps:
    model = get_best_trained_model(hp)
    model.evaluate(x_test, y_test)
    best_models.append(model)
```

Если вас не беспокоит некоторое отставание в качестве модели, можно пойти по более короткому пути: просто используйте тюнер для загрузки наиболее эффективной модели с лучшими весами, сохраненными во время поиска гиперпараметров, без повторного обучения новых моделей с нуля:

```
best_models = tuner.get_best_models(top_n)
```

### **ПРИМЕЧАНИЕ**

При использовании автоматического оптимизатора гиперпараметров важно учитывать одну серьезную проблему — переобучение на проверочном наборе. Поскольку настройка гиперпараметров выполняется по результатам оценки на проверочных данных, их обучение происходит на проверочном наборе и, как следствие, быстро наступает эффект их переобучения. Всегда помните об этом.

### **Искусство создания правильного пространства поиска**

В целом оптимизация гиперпараметров — мощный метод, который абсолютно необходим 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, 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 принципиальным образом оптимизировать их.

Но имейте в виду, что оптимизация гиперпараметров не отменяет необходимости знакомства с передовыми архитектурами моделей. Пространства поиска растут в геометрической прогрессии с количеством вариантов, поэтому было бы слишком дорого превращать все и вся в гиперпараметры и перекладывать задачу их подбора на автоматический механизм оптимизации. Следует с умом

подходить к конструированию правильного пространства поиска. Настройка гиперпараметров — это автоматизация, а не волшебство: вы используете ее для автоматического проведения экспериментов, которые в ином случае вам пришлось бы выполнять вручную. Тем не менее автоматизация не избавляет от необходимости самому подбирать возможные конфигурации экспериментов, которые могут дать хорошие показатели.

Однако у меня для вас есть хорошая новость: используя механизм оптимизации гиперпараметров, решения о выборе конфигурации, которые вы должны принимать, переходят из плоскости микрорешений (таких как выбор количества обучаемых параметров в том или ином слое) в плоскость архитектурных решений более высокого уровня (таких как необходимость использования остаточных связей в модели). Микрорешения специфичны для определенной модели и определенного набора данных; высокоуровневые же архитектурные решения можно обобщить для разных задач и наборов данных. Например, практически любую задачу классификации изображений можно решить с помощью одного и того же шаблона пространства поиска.

Следуя этой логике, KerasTuner предлагает *заранее подготовленные пространства поиска* для широких категорий задач (таких как классификация изображений). Благодаря им вы можете просто добавить данные, запустить поиск и получить довольно хорошую модель. Попробуйте гипермодели `kt.applications.HyperXception` и `kt.applications.HyperResNet`, которые являются эффективно настраиваемыми версиями моделей Keras Applications.

## **Будущее оптимизации гиперпараметров: автоматизированное машинное обучение**

В настоящее время работа инженера, занимающегося глубоким обучением, заключается в основном в обработке данных с помощью сценариев на Python и дальнейшей тщательной настройке архитектуры и гиперпараметров глубокой сети для получения рабочей модели — или даже суперсовременной модели, если инженер достаточно честолюбив. Нет нужды говорить, что это неоптимальный подход. Однако автоматизация может помочь делу.

Поиск по набору возможных скоростей обучения или размеров слоев — это только первый шаг. Можно пойти дальше и попытаться разработать подходящую архитектуру с нуля с минимально возможными ограничениями: например, используя обучение с подкреплением или генетические алгоритмы. В будущем появятся целые сквозные конвейеры машинного обучения, которые будут генерироваться автоматически, а не создаваться инженерами-ремесленниками вручную. Это называется автоматическим машинным обучением, или *AutoML*. Вы уже можете использовать такие библиотеки, как AutoKeras (<https://github.com/keras-team/autokeras>), для решения простых задач машинного обучения с минимальным участием с вашей стороны.

На сегодняшний день AutoML все еще находится в зачаточном состоянии и не подходит для решения масштабных задач. Но, когда автоматическое машинное обучение станет достаточно зрелым для широкого внедрения, инженеры глубокого обучения не лишатся работы — они займутся цепочками создания ценностей. Они будут прикладывать больше усилий для разработки сложных функций потерять, точнее отражающих бизнес-цели, и для 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.

Всегда смотрите на общую картину, сосредоточьтесь на понимании основ и имейте в виду, что узкоспециализированная рутинная в конечном итоге будет автоматизирована. Воспринимайте это как подарок — возможность повысить производительность ваших рабочих процессов, — а не как угрозу собственной значимости. Ваша работа не должна превращаться в бесконечную настройку всего и вся.

### **13.1.2. 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 (определеняемых 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 полной 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)
```

Использовать четыре разные модели 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 случайногопоиска или простой оптимизации, такой как алгоритм Нелдера — Мида (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, например вы можете вычислить среднее экспоненциальное прогнозов. 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 отличаются 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 и я вместе заняли четвертое место в решении 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.

## 13.2. МАСШТАБИРОВАНИЕ ОБУЧЕНИЯ МОДЕЛЕЙ

Вспомните концепцию «цикличности движения вперед», представленную в главе 7: качество ваших идей зависит от того, сколько циклов уточнения они прошли (рис. 13.1). А скорость, с которой вы сможете развивать эту идею, зависит от того, насколько быстро вы станете настраивать новые эксперименты, проводить эти эксперименты и, наконец, насколько хорошо вы проанализируете полученные данные.

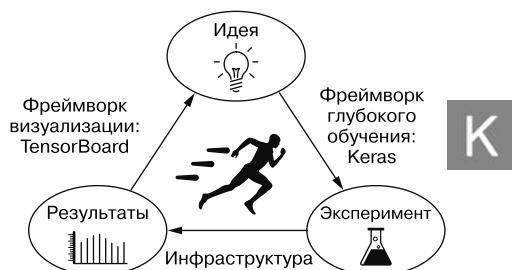


Рис. 13.1. Циклическое движение вперед

По мере обретения опыта использования Keras API скорость подготовки новых экспериментов по глубокому обучению перестанет быть для вас узким местом в этом циклическом движении вперед, но останется еще одно — скорость обучения моделей. Быстрой можно назвать инфраструктуру обучения, если она позволяет получить результаты через 10–15 минут — благодаря этому вы сможете выполнять десятки итераций каждый день. А чем быстрее обучение, тем выше *качество* ваших решений.

В этом разделе вы познакомитесь с тремя способами ускорения обучения моделей:

- с обучением со смешанной точностью, которое можно использовать даже с GPU;
- с обучением на нескольких GPU;
- с обучением на TPU.

Поехали.

### **13.2.1. Ускорение обучения на GPU со смешанной точностью**

Что, если я скажу вам, что есть простой способ ускорить обучение почти любой модели в три раза и практически бесплатно? Многим это покажется слишком хорошим, чтобы быть правдой, — и все же такой способ действительно существует. Это *обучение со смешанной точностью*. Чтобы понять, как это возможно, сначала нужно рассмотреть понятие точности в информатике.

#### **Точность представления вещественных чисел**

Точность для чисел — то же самое, что разрешение для изображений. Компьютер может обрабатывать только единицы и нули, поэтому любое число, которое он видит, должно быть представлено в виде последовательности нулей и единиц. Многие из вас знакомы с целыми числами `uint8`, которые отражают целочисленные значения, закодированные восемью битами: `00000000` представляет 0, а `11111111` — 255. Чтобы закодировать целые числа больше 255, нужно добавить больше битов — восьми будет недостаточно. Большинство целых чисел хранятся в 32 битах, с помощью которых можно представлять целые числа со знаком в диапазоне от  $-2\ 147\ 483\ 648$  до  $2\ 147\ 483\ 647$ .

То же относится к вещественным числам (числам с плавающей запятой). В математике они образуют непрерывную последовательность: между любыми двумя числами находится бесконечное количество точек, и вы всегда можете увеличить масштаб оси вещественных чисел. Однако в информатике все совсем иначе: например между 3 и 4 существует конечное число промежуточных точек. Как много? Зависит от *точности* используемого представления — количества битов, используемых для хранения числа. Это количество можно увеличивать только до определенного предела.

Существует три уровня точности, которые обычно используются в информатике:

- половинная точность, или `float16`, когда числа хранятся в 16 битах;
- одинарная точность, или `float32`, когда числа хранятся в 32 битах;
- двойная точность, или `float64`, когда числа хранятся в 64 битах.

#### ПРИМЕЧАНИЕ О ПРЕДСТАВЛЕНИИ ВЕЩЕСТВЕННЫХ ЧИСЕЛ В ФОРМАТЕ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Представление вещественных чисел в формате с плавающей точкой вопреки здравому смыслу имеет неравномерное распределение. Большие числа имеют меньшую точность: между  $2^{N}$  и  $2^{N+1}$  столько же представимых значений, сколько между 1 и 2, для любого  $N$ .

Данный факт связан с тем, что числа в представлении с плавающей точкой кодируются тремя частями — знаком, значением (его еще называют мантиссой) и показателем степени:

```
{знак} * (2 ** ({показатель степени} - 127)) * 1.{мантилса}
```

Например, вот как можно закодировать значение `float32`, ближайшее к числу пи:



```
value = +1 * (2 ** (128 - 127)) * 1.5707963705062866
value = 3.1415927410125732
```

Число пи в представлении с плавающей точкой с одинарной точностью, включающем бит знака, целочисленный показатель степени и целочисленную мантиссу

По этой причине числовая ошибка, возникающая при преобразовании числа в его представление с плавающей точкой, может сильно различаться в зависимости от точного значения, и эта ошибка имеет тенденцию увеличиваться для чисел с большим абсолютным значением.

Разрешение чисел с плавающей точкой можно рассматривать как наименьшее расстояние между двумя произвольными числами, которое можно представить. Для представлений с одинарной точностью это около  $1\text{e}-7$ . С двойной точностью — около  $1\text{e}-16$ . С половинной точностью — всего лишь  $1\text{e}-3$ .

Все модели, которые вы видели в этой книге, использовали числа с одинарной точностью: они сохраняли свое состояние в переменных типа `float32` и выполняли вычисления с входными данными типа `float32`. Этой точности достаточно для прямого и обратного проходов модели без потери какой-либо информации, особенно если речь идет о небольших изменениях градиента (напомню, что типичная скорость обучения составляет  $1\text{e}-3$ ; довольно часто можно увидеть и изменения весов порядка  $1\text{e}-6$ ).

Также можно использовать числа типа `float64`, хотя это и избыточно: такие операции, как умножение или сложение матриц чисел с двойной точностью, обходятся намного дороже, а модель будет выполнять вдвое больше работы без каких-либо явных преимуществ. Но то же самое нельзя сделать с весами `float16`: процесс градиентного спуска не будет гладким из-за невозможности представить небольшие изменения градиента — около  $1\text{e}-5$  или  $1\text{e}-6$ .

### ОСТЕРЕГАЙТЕСЬ ЗНАЧЕНИЙ ПО УМОЛЧАНИЮ ДЛЯ DTYPES

В Keras и TensorFlow по умолчанию берется одинарная точность: любые созданные вами тензор или переменная, если явно не указать иное, будут иметь тип `float32`. Однако для массивов NumPy по умолчанию используется тип `float64`!

Преобразование массива NumPy с типом по умолчанию в тензор TensorFlow даст в результате тензор `float64`, который, возможно, вам совсем не нужен:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> np_array = np.zeros((2, 2))
>>> tf_tensor = tf.convert_to_tensor(np_array)
>>> tf_tensor.dtype
tf.float64
```

Не забывайте явно указывать типы данных при преобразовании массивов NumPy:

```
>>> np_array = np.zeros((2, 2))
>>> tf_tensor = tf.convert_to_tensor(np_array, dtype="float32") ←
>>> tf_tensor.dtype
tf.float32
```

Явно указывайте тип  
через параметр `dtype`

Обратите внимание, что, если вызвать метод `fit()` из библиотеки Keras с данными в массиве NumPy, он выполнит это преобразование автоматически.

Однако можно использовать гибридный подход — в этом и заключается суть обучения со смешанной точностью. Идея следующая: применять 16-битные вычисления там, где не нужна высокая точность, и 32-битные — в других местах, чтобы поддержать числовую стабильность. Современные GPU и TPU оснащены специализированными аппаратными модулями, способными выполнять 16-битные операции намного быстрее и использовать меньше памяти, чем при выполнении эквивалентных 32-битных операций. Используя низкоточные операции там, где это допустимо, можно значительно ускорить обучение на таких устройствах. А обеспечивая вычисления с одинарной точностью в частях модели, чувствительных к точности, можно получить ряд преимуществ без существенного влияния на качество модели.

И эти преимущества значительны: на современных графических процессорах NVIDIA смешанная точность может ускорить обучение до трех раз. Данный прием может также пригодиться при обучении на TPU (мы обсудим этот вопрос чуть ниже), где обучение может быть ускорено до 60 %.

## Обучение со смешанной точностью на практике

Вот как можно включить использование смешанной точности при обучении на GPU:

```
from tensorflow import keras
keras.mixed_precision.set_global_policy("mixed_float16")
```

Большая часть прямого прохода модели в этом случае будет выполняться с использованием чисел `float16` (за исключением численно нестабильных операций, таких как `softmax`), а веса модели будут храниться и изменяться в `float32`.

Слои Keras имеют атрибуты `variable_dtype` и `compute_dtype`. По умолчанию оба имеют значение `float32`. При включении режима смешанной точности атрибуту `compute_dtype` большинства слоев присваивается значение `float16`; в результате эти слои будут приводить входные данные к типу `float16` и выполнять вычисления в `float16` (используя копии весов с половинной точностью). Однако, поскольку их атрибут `variable_dtype` по-прежнему имеет значение `float32`, веса будут получать от оптимизатора точные изменения с плавающей точкой.

Обратите внимание на то, что при использовании типа `float16` некоторые операции могут быть численно нестабильными (в частности, `softmax` и `crossentropy`). Если вам понадобится отказаться от смешанной точности в определенном слое, просто передайте аргумент `dtype="float32"` конструктору этого слоя.

### 13.2.2. Обучение на нескольких GPU

В то время как графические процессоры с каждым годом становятся все мощнее, модели глубокого обучения становятся все больше и требуют все больше вычислительных ресурсов. Обучение на одном графическом процессоре жестко ограничивает вашу возможную скорость. Есть ли решение? Вы можете просто добавить больше GPU и перейти на *распределенное обучение на нескольких GPU*.

Есть два способа распределения вычислений между несколькими устройствами: *параллелизм данных* и *параллелизм моделей*.

При параллелизме данных одна модель копируется на несколько устройств. Каждая копия модели обрабатывает разные пакеты данных, а затем полученные результаты объединяются.

При параллелизме моделей разные части одной модели работают на разных устройствах, одновременно обрабатывая один пакет данных. Этот способ лучше подходит для моделей, имеющих естественную параллельную архитектуру, например для моделей с несколькими ветвями.

На практике параллелизм моделей используется только для моделей, которые слишком велики, чтобы уместиться на одном устройстве, — и не как способ ускорения обучения обычных моделей, а как способ обучения очень больших моделей. В этой книге мы не будем его рассматривать, а сосредоточимся на более распространенном параллелизме данных. Давайте посмотрим, как этот способ реализовать.

#### Приобретение двух или более графических процессоров

Прежде всего вам нужно получить доступ к нескольким GPU. В настоящее время Google Colab позволяет использовать только один GPU, поэтому придется сделать одно из двух:

- приобрести 2–4 GPU, смонтировать их на одной машине (для этого потребуется мощный блок питания) и установить драйверы CUDA, cuDNN и т. д. Но для большинства из нас это не лучший вариант;
- арендовать виртуальную машину с несколькими GPU в Google Cloud, Azure или AWS. Вы сможете использовать образы виртуальных машин с предустановленными драйверами и программным обеспечением, благодаря чему затраты на настройку будут минимальными. Вероятно, это лучший вариант для тех, кто не занимается обучением моделей круглосуточно и без выходных.

Я не буду подробно рассказывать о том, как развернуть облачные экземпляры с несколькими GPU: техника быстро развивается, поэтому такие инструкции относительно недолговечны; плюс вся необходимая информация легко доступна в интернете.

А если вы не хотите связываться с накладными расходами на управление собственными экземплярами виртуальных машин, то можете использовать TensorFlow Cloud (<https://github.com/tensorflow/cloud>) — пакет, который я и моя команда выпустили недавно. Он позволит вам начать обучение на нескольких GPU, просто добавив одну строку кода в начало блокнота Colab. Если вы хотите безболезненно перейти от отладки модели в Colab к максимально быстрому ее обучению на любом количестве GPU, ознакомьтесь с этим пакетом.

## Синхронное обучение на одном хосте с несколькими устройствами

Как только у вас появится машина с несколькими GPU, на которой вы сможете выполнить код `import tensorflow`, вы окажетесь очень близко к обучению распределенной модели. Вот как это организовать:

```
Создать объект распределенной стратегии.  
MirroredStrategy — лучшее решение  
  
strategy = tf.distribute.MirroredStrategy() ←  
print(f"Number of devices: {strategy.num_replicas_in_sync}")  
with strategy.scope(): ← Открыть контекст стратегии  
    model = get_compiled_model()  
model.fit(  
    train_dataset,  
    epochs=100,  
    validation_data=val_dataset,  
    callbacks=callbacks)
```

Все, что создает переменные, должно действовать в контексте стратегии. В общем случае в контексте стратегии достаточно создать модель и скомпилировать ее

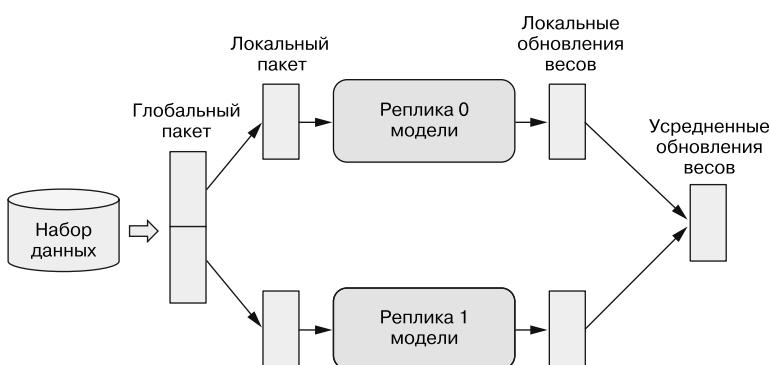
Обучить модель на всех доступных устройствах

Эти несколько строк реализуют наиболее распространенную конфигурацию обучения: *синхронное обучение на одном хосте с несколькими устройствами*, известную в TensorFlow как стратегия зеркального распределения. Под «одним хостом» подразумевается, что все GPU находятся на одной машине (в отличие от кластера из множества машин, каждая со своим GPU, взаимодействующих по сети). «Синхронное обучение» означает, что состояние копий модели для каждого GPU всегда остается неизменным, но есть варианты распределенного обучения, где это не так.

Когда вы открываете контекст `MirroredStrategy` и конструируете в нем свою модель, объект `MirroredStrategy` создает одну копию модели (реплику) на каждом доступном GPU. Затем каждый этап обучения разворачивается следующим образом (рис. 13.2).

1. Из набора данных извлекается пакет (называемый *глобальным пакетом*).
2. Он разбивается на четыре пакета меньшего размера (называемых *локальными пакетами*). Например, если в глобальном пакете 512 образцов, то в каждом из четырех локальных пакетов будет 128 образцов. Для оптимальной загрузки GPU желательно, чтобы локальные пакеты были достаточно большими, соответственно, глобальному пакету обычно хорошо быть очень большим.

3. Каждая из четырех реплик независимо обрабатывает один локальный пакет на своем устройстве: они выполняют прямой, а затем обратный проход. Каждая реплика выводит «дельту веса», описывающую величину обновления каждой весовой переменной в модели с учетом градиента предыдущих весов по отношению к потерям модели в локальном пакете.
4. Дельты весов, возникающие из локальных градиентов в четырех репликах, эффективно объединяются в глобальную дельту, которая применяется ко всем репликам. Поскольку это делается в конце каждого шага, реплики всегда остаются синхронизированными: их веса всегда равны.



**Рис. 13.2.** Один шаг обучения MirroredStrategy: каждая реплика модели вычисляет локальные обновления весов, которые затем объединяются и используются для обновления состояния всех реплик

#### СОВЕТЫ ПО УВЕЛИЧЕНИЮ ПРОИЗВОДИТЕЛЬНОСТИ TF.DATA

При распределенном обучении всегда предоставляйте данные в виде объекта `tf.data.Dataset`, чтобы гарантировать наилучшую производительность. (Также можно передавать данные в виде массивов NumPy, потому что они автоматически преобразуются в объекты `Dataset` внутри метода `fit()`.) Кроме этого, желательно использовать опережающую выборку данных: перед передачей набора данных в `fit()` вызовите `dataset.prefetch(buffer_size)`. Если вы не уверены в размере буфера, передайте параметр `dataset.prefetch(tf.data.AUTOTUNE)`, который выберет размер буфера за вас.

В идеальном мире обучение на  $N$  графических процессорах привело бы к ускорению в  $N$  раз. Однако в реальной жизни механизм распределения добавляет свои накладные расходы — в частности, объединение приращений весов,

поступающих с разных устройств, занимает некоторое время. Эффективное ускорение, которое получится в конечном итоге, зависит от количества используемых GPU:

- с двумя GPU ускорение получится близким к двукратному;
- с четырьмя составит что-то около 3,8 раза;
- с восемью — примерно 7,3 раза.

При этом предполагается, что у вас есть глобальный пакет достаточно большого размера, чтобы каждый GPU использовался на полную мощность. Если размер пакета слишком мал, локального пакета будет недостаточно, чтобы загрузить работой ваши GPU.

### 13.2.3. Обучение на TPU

Помимо графических процессоров, в мире глубокого обучения все чаще отдают предпочтение более специализированному оборудованию, разработанному специально для рабочих процессов глубокого обучения. Такие одноцелевые микросхемы известны как специализированные интегральные схемы (Application-Specific Integrated Circuits, ASIC). Разные компании, большие и малые, работают над созданием новых микросхем, но наибольшего успеха в этом направлении в настоящее время добились в Google с их тензорными процессорами (Tensor Processing Unit, TPU), доступными в Google Cloud и Google Colab.

Обучение на TPU сопряжено с определенными сложностями, но оно того стоит: TPU действуют очень, очень быстро. Обучение на TPU V2 обычно происходит в 15 раз быстрее, чем на GPU NVIDIA P100. Для большинства моделей обучение на TPU оказывается в среднем в три раза экономичнее, чем на GPU.

#### Использование TPU в Google Colab

Фактически в Colab можно бесплатно использовать 8-ядерный TPU. В меню Colab Runtime (Среда выполнения) выберите пункт Change Runtime Type (Сменить среду выполнения) — и в открывшемся диалоге, в раскрывающемся списке Hardware Accelerator (Аппаратный ускоритель), вы увидите, что, помимо среды выполнения GPU, вам доступна среда выполнения TPU.

При использовании среды выполнения GPU ваши модели будут брать GPU напрямую — вам не придется что-то дополнительно настраивать. Но со средой выполнения TPU ситуация иная — вы должны сделать дополнительный шаг, прежде чем начать создавать модели, а именно подключиться к кластеру TPU.

Вот как это делается:

```
import tensorflow as tf
tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect()
print("Device:", tpu.master())
```

Не обязательно досконально понимать, что делает данный код, — это всего лишь короткое заклинание, соединяющее среду выполнения вашего ноутбука с устройством. Сезам, открайся!

Как и при обучении на нескольких GPU, использование TPU требует открыть контекст стратегии распределения — в данном случае `TPUStrategy`. Стратегия `TPUStrategy` следует тому же шаблону распределения, что и `MirroredStrategy`: для каждого ядра TPU создается своя реплика модели, и все реплики синхронизируются.

Вот простой пример.

#### Листинг 13.4. Создание модели в контексте `TPUStrategy`

```
from tensorflow import keras
from tensorflow.keras import layers

strategy = tf.distribute.TPUStrategy(tpu)
print(f"Number of replicas: {strategy.num_replicas_in_sync}")

def build_model(input_size):
    inputs = keras.Input((input_size, input_size, 3))
    x = keras.applications.resnet.preprocess_input(inputs)
    x = keras.applications.resnet.ResNet50(
        weights=None, include_top=False, pooling="max")(x)
    outputs = layers.Dense(10, activation="softmax")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

with strategy.scope():
    model = build_model(input_size=32)
```

Теперь можно начинать обучение. Но есть одна любопытная особенность обучения на TPU в Colab: конфигурация с двумя виртуальными машинами. Виртуальная машина со средой выполнения вашего блокнота — это отдельная виртуальная машина, отличная от виртуальной машины, оснащенной TPU. Поэтому не получится просто запустить обучение на данных из файлов, хранящихся

на локальном диске (то есть на диске, связанном с виртуальной машиной, где выполняется блокнот). Среда выполнения TPU не может читать данные оттуда. Поэтому вам придется эти данные загрузить. Сделать это можно двумя способами:

- провести обучение на данных, находящихся в памяти виртуальной машины (не на диске). Если ваши данные располагаются в массиве NumPy, то ничего больше делать не нужно;
- сохранить данные в корзину Google Cloud Storage (GCS) и создать объект набора данных, который будет читать данные непосредственно из корзины без загрузки на локальную машину. Среда выполнения TPU может читать данные из GCS. Это единственный вариант для наборов данных, слишком больших, чтобы уместиться в памяти.

Давайте выполним обучение на данных, находящихся в массивах NumPy в памяти, — наборе данных CIFAR10:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()  
model.fit(x_train, y_train, batch_size=1024)
```

Обратите внимание, что обучение на TPU производится практически так же, как обучение на GPU, — нужно создать пакеты больших размеров, чтобы обеспечить максимальную загрузку устройства

Вы заметите, что первая эпоха выполнится с некоторой задержкой: для компиляции модели в нечто, что может выполнить TPU, необходимо время. Остальное обучение протекает молниеносно.

### ВВОД/ВЫВОД КАК УЗКОЕ МЕСТО

TPU могут очень быстро обрабатывать пакеты данных, поэтому скорость чтения данных из корзины GCS легко может стать узким местом.

- Если набор данных достаточно мал, сохраните его в памяти виртуальной машины. Это можно сделать вызовом метода `dataset.cache()` объекта набора данных. В этом случае данные будут читаться из GCS только один раз.
- Если набор данных слишком велик и не умещается в памяти, сохраните его в виде файлов TFRecord — эффективном двоичном формате, который обеспечивает возможность быстрой загрузки. На сайте keras.io вы найдете пример, демонстрирующий сохранение данных в файлах TFRecord ([https://keras.io/examples/keras\\_recipes/creating\\_tfrecords/](https://keras.io/examples/keras_recipes/creating_tfrecords/)).

### Использование слияния этапов для оптимизации загрузки TPU

Устройства TPU обладают значительной вычислительной мощностью, поэтому вам придется работать с очень большими пакетами, чтобы обеспечить максимальную загрузку ядер TPU. Для небольших моделей оптимальный размер пакета может быть огромным — до 10 000 образцов. Соответственно, необходимо будет увеличить скорость обучения оптимизатора; обновлений весов будет меньше, но каждое обновление будет более точным (потому что градиенты вычисляются с использованием большего количества точек данных). Поэтому желательно перемещать веса на большую величину с каждым обновлением.

Тем не менее есть возможность сохранить разумный размер пакетов при оптимальной загрузке TPU, прибегнув к простому трюку — *слиянию этапов*. Идея состоит в том, чтобы выполнить несколько этапов обучения за один сеанс выполнения на TPU, то есть сделать больше работы между двумя операциями передачи данных из памяти виртуальной машины в TPU. Для этого просто передайте аргумент `steps_per_execution` в вызов метода `compile()`, например `steps_per_execution=8`, чтобы выполнить восемь этапов обучения за один сеанс выполнения на TPU. Для небольших моделей, использующих мощности TPU недостаточно оптимально, это может привести к значительному ускорению.

## КРАТКИЕ ИТОГИ ГЛАВЫ

- Для автоматизации утомительного поиска наилучшей конфигурации модели можно использовать настройку гиперпараметров и KerasTuner. Но помните об опасности переобучения на проверочном наборе данных!
- Ансамбль разнообразных моделей часто может значительно улучшить качество прогнозов.
- Ускорить обучение модели на GPU можно, использовав прием обучения со смешанной точностью — обычно он дает хороший прирост скорости без дополнительных усилий с вашей стороны.
- Для дальнейшего масштабирования рабочих процессов можно применять стратегию `tf.distribute.MirroredStrategy` и с ее помощью обучать модели на нескольких GPU.
- У вас даже есть возможность потренироваться в обучении на устройствах Google TPU (доступны в Colab), применив стратегию `TPUStrategy`. Если модель небольшая, обязательно используйте слияние этапов обучения (настраивается с помощью аргумента `compile(..., steps_per_execution=N)`), чтобы максимально загрузить ядра TPU работой.

# 11

## Заключение

### В этой главе

- ✓ 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.

## 14.1. КРАТКИЙ ОБЗОР КЛЮЧЕВЫХ ПОНЯТИЙ

Данный раздел обобщает ключевые выводы из этой книги. Если вам потребуется быстро освежить в памяти все, что вы изучили здесь, прочитайте следующие несколько страниц.

### 14.1.1. 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 годах, а доминирующей она стала только в 2000-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*: 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, минимизирующих *функцию потерь*. Поскольку цепочка геометрических преобразований является дифференцируемой, обновление весов с целью минимизации функции потерь эффективно выполняется с помощью *градиентного спуска*.

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.

### **14.1.2. Что делает глубокое обучение особенным среди других подходов к машинному обучению**

В течение всего лишь нескольких лет глубокое обучение добилось 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 другое. При наличии достаточного объема обучающих данных (например, обучающих данных, предварительно классифицированных людьми) из сенсорной информации с помощью глубокого обучения 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 «лета». Завершится ли данный период в ближайшем будущем и что случится в конце — 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 нескольких десятилетий.

### **14.1.3. Как правильно воспринимать глубокое обучение**

Самое удивительное в глубоком обучении — простота реализации. Еще десять лет 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: все, что

нам нужно, — это достаточно большие параметрические модели, обученные методом градиентного спуска на достаточно большом количестве примеров. Как однажды сказал Ричард Фейнман о Вселенной, «она не сложная, просто очень большая»<sup>1</sup>.

В глубоком обучении все 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, которые итеративно обновляются, в зависимости от качества работы модели. Ключевой характеристикой такого геометрического преобразования является *дифференцируемость*, которая совершенно необходима 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. Все, что вам нужно, — это пространства с достаточно большой размерностью, чтобы полностью охватить отношения, присутствующие в исходных данных.

Все основано на одной главной идеи: смысл *заключается в попарных отношениях* (между словами в языке, между пикселями 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 нетрудно представить применение других структур данных

---

<sup>1</sup> Интервью с Ричардом Фейнманом, *The World from Another Point of View*, телевидение Йоркшира, 1972.

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.

#### **14.1.4. Ключевые технологии**

Технологическая революция, разворачивающаяся на наших глазах, начиналась не с какого-то одного прорывного изобретения. Как любая другая революция, она явилась 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 Хуан 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 выполняющего дифференцирование, и Keras, обеспечивающего доступность глубокого обучения для широких масс.

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. Всеми, 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.

#### **14.1.5. Обобщенный процесс машинного обучения**

Хорошо иметь доступ к чрезвычайно мощному инструменту создания моделей, отображающих любое входное пространство в любое целевое, однако не менее сложной частью процесса машинного обучения является все 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 6.

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, покрывающий ваши потребности) и только потом добавлять регуляризацию или уменьшать размер модели. 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!
7. Разверните получившуюся модель в промышленном окружении — как веб-API, как часть приложения на JavaScript или C++, на встроенном устройстве и т. д. Продолжайте следить за качеством ее работы на реальных данных и используйте полученные результаты для уточнения следующего поколения модели!

### 14.1.6. Основные архитектуры сетей

Есть три семейства архитектур сетей, которые вы должны знать: *полносвязные, сверточные и рекуррентные сети*, а также архитектура Transformer. Каждый тип моделей предназначен для конкретной модальности входных данных. Архитектура сети 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 мультиmodalных моделей, подобно деталям конструктора лего. В некотором смысле слои глубокого обучения — это детали лего 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.
- *Последовательные данные* — рекуррентные сети для временных рядов или архитектура Transformer для дискретных последовательностей (таких как последовательности слов). Одномерные сверточные сети можно использовать для непрерывных последовательностей, инвариантных в отношении переноса, например сигналов волновой формы.
- *Видеоданные* — трехмерные сверточные сети (если необходимо захватывать эффекты движения) или комбинация двумерной сверточной сети,

действующей на уровне кадров для извлечения признаков, с последующей обработкой моделью последовательностей.

- *Объемные данные* — трехмерные сверточные сети.

Теперь вспомним особенности каждой архитектуры.

### Полносвязные сети

Полносвязная сеть — это стек слоев `Dense`, предназначенных для обработки векторных данных (где каждый образец представлен вектором количественных или качественных атрибутов). 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 4. Они также применяются для заключительной классификации или регрессии в большинстве сетей. Например, сверточные сети, рассматривавшиеся в главе 8, а также рекуррентные сети, рассматривавшиеся в главе 10, 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 tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
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`:

```
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
```

```
outputs = layers.Dense(num_classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)
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 многомерным прямым кодированием:

```
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_classes, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
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):

```
inputs = keras.Input(shape=(num_input_features,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_values)(x)
model = keras.Model(inputs, outputs)
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, слой **Conv2D** — для обработки изображений и слой **Conv3D** — для обработки объемов. В качестве более компактной и эффективной альтернативы сверточным слоям также можно использовать свертки, разделяемые по глубине, такие как **SeparableConv2D**.

*Сверточные нейронные сети* состоят из стека сверточных слоев и слоев выбора максимальных значений по соседям (max-pooling). Слои выбора позволяют снижать пространственную размерность данных, что необходимо для

сохранения размеров карты признаков в разумных пределах с ростом числа признаков, и дают 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.

Вот типичная сеть для классификации изображений (в данном случае категориальная классификация), использующая слои `SeparableConv2D`:

```
inputs = keras.Input(shape=(height, width, channels))
x = layers.SeparableConv2D(32, 3, activation="relu")(inputs)
x = layers.SeparableConv2D(64, 3, activation="relu")(x)
x = layers.MaxPooling2D(2)(x)
x = layers.SeparableConv2D(64, 3, activation="relu")(x)
x = layers.SeparableConv2D(128, 3, activation="relu")(x)
x = layers.MaxPooling2D(2)(x)
x = layers.SeparableConv2D(64, 3, activation="relu")(x)
x = layers.SeparableConv2D(128, 3, activation="relu")(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="categorical_crossentropy")
```

В очень глубокие сверточные сети часто добавляются слои *пакетной нормализации*, а также *остаточные связи* — два архитектурных шаблона, которые помогают градиентной информации беспрепятственно проходить через сеть.

## Рекуррентные нейронные сети

*Рекуррентные нейронные сети* 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 неинвариантны 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 случаев практического применения 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, то сеть возвращает только последний вывод, содержащий информацию обо всей последовательности.

Вот единственный простой слой RNN для бинарной классификации последовательностей векторов:

```
inputs = keras.Input(shape=(num_timesteps, num_features))
x = layers.LSTM(32)(inputs)
outputs = layers.Dense(num_classes, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")
```

А вот стек из слоев RNN для бинарной классификации последовательностей векторов:

```
inputs = keras.Input(shape=(num_timesteps, num_features))
x = layers.LSTM(32, return_sequences=True)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
x = layers.LSTM(32)(x)
outputs = layers.Dense(num_classes, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")
```

## Архитектура Transformer

Модель с архитектурой Transformer просматривает набор векторов (например, векторов слов) и применяет механизм *нейронного внимания* для преобразования каждого вектора в представление, учитывающее *контекст*, который формируется другими векторами в наборе. Когда рассматриваемый набор является упорядоченной последовательностью, можно также использовать *позиционное кодирование* и создавать модели Transformer, способные учитывать глобальный контекст и порядок слов и обрабатывать длинные текстовые абзацы гораздо эффективнее, чем рекуррентные или одномерные сверточные сети.

Модель Transformer можно использовать для решения любых задач, связанных с обработкой наборов или последовательностей, включая классификацию текста, но лучше всего она подходит для задач *обучения типа «последовательность в последовательность»*, таких как машинный перевод абзацев с одного языка на другой.

Модель Transformer для преобразования последовательностей в последовательности состоит из двух частей:

- **TransformerEncoder** — кодировщик, преобразующий входную последовательность векторов в контекстно зависимую последовательность выходных векторов с учетом порядка;
- **TransformerDecoder** — декодер, принимающий выходные данные **TransformerEncoder** и целевую последовательность и предсказывающий, что должно находиться в целевой последовательности дальше.

Для обработки единственной последовательности (или набора) векторов используется только `TransformerEncoder`.

Ниже приводится пример модели Transformer для преобразования последовательности в последовательность, которая отображает исходную последовательность в целевую (эта конфигурация может использоваться, например, для машинного перевода или выбора ответов на вопросы):

```

    Исходная последовательность
encoder_inputs = keras.Input(shape=(sequence_length,), dtype="int64") ←
x = PositionalEmbedding(
    sequence_length, vocab_size, embed_dim)(encoder_inputs) ← Целевая последовательность
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)   в настоящий момент
decoder_inputs = keras.Input(shape=(None,), dtype="int64") ←
x = PositionalEmbedding(
    sequence_length, vocab_size, embed_dim)(decoder_inputs) ← Целевая последовательность
                                                на один шаг вперед в будущем
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x) ←
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
transformer.compile(optimizer="rmsprop", loss="categorical_crossentropy")

```

А вот одиночный кодировщик `TransformerEncoder` для бинарной классификации целочисленных последовательностей:

```

inputs = keras.Input(shape=(sequence_length,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop", loss="binary_crossentropy")

```

Полные реализации слоев `TransformerEncoder`, `TransformerDecoder` и `PositionalEmbedding` представлены в главе 11.

### 14.1.7. Пространство возможностей

Что можно построить, используя приемы глубокого обучения? Помните, что конструирование моделей глубокого обучения напоминает игру с конструктором лего: 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 демонстрируются несколько примеров, чтобы показать, что глубокое обучение позволяет решать не только задачи классификации и регрессии, которые традиционно были хлебом насущным 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 с 14.2 по 14.4 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, а также управление акселератором и тормозом;
  - *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:
  - *машинный перевод* — отображение абзаца текста на одном языке в перевод на другом языке;

- *интеллектуальный автоответчик* — генерирование однострочных ответов на электронные письма;
  - *ответы на вопросы* — генерирование ответов на общие вопросы;
  - *резюмирование* — преобразование длинных статей в краткие обзоры.
- Отображение изображений в текст:
    - *распознавание текста* — преобразование текста на изображении в соответствующую текстовую строку;
    - *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 все*, но *не совсем все*. Давайте в следующем разделе посмотрим, чего *нельзя* сделать с помощью глубокого обучения.

## 14.2. 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.

### 14.2.1. 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. 14.1).



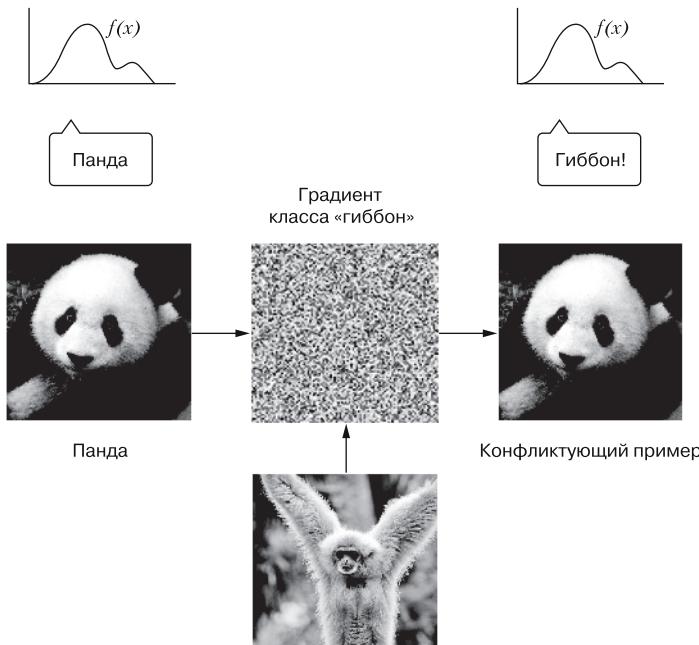
Мальчик, держащий бейсбольную биту

**Рис. 14.1.** Ошибка системы создания подписей к изображениям, основанной на глубоком обучении

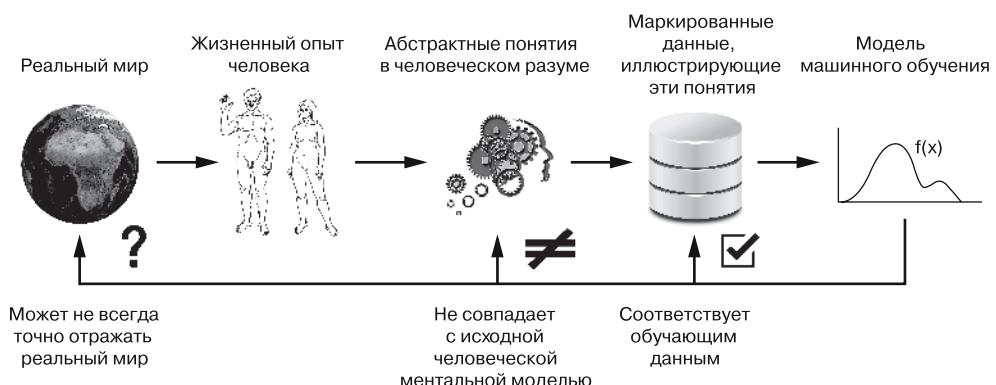
Это особенно ярко подчеркивается *примерами с состязательными сетями*, когда в сеть глубокого обучения передаются образцы, сконструированные специально для того, чтобы обмануть модель. Вы уже знаете, что можно, например, выполнить градиентное восхождение во входном пространстве и сгенерировать входные данные, максимизирующие функцию активации некоторого сверточного фильтра, — это основа приема визуализации фильтров, представленного в главе 9, а также алгоритма DeepDream, который мы обсуждали в главе 12. 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. 14.2). 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 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. 14.3). Модели, создаваемые вами, будут использовать любую возможность для минимизации потерь и максимального приближения к обучающим данным. Например, модели

изображений, как правило, больше полагаются на локальные текстуры, чем на глобальное понимание входных изображений, поэтому модель, обученная на наборе данных, включающем фотографии леопардов и диванов, почти наверняка классифицирует леопардовый диван как настоящего леопарда.



**Рис. 14.2.** Незаметные изменения в изображении могут мешать модели правильно его классифицировать



**Рис. 14.3.** 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.

### 14.2.2. Автоматы и носители интеллекта

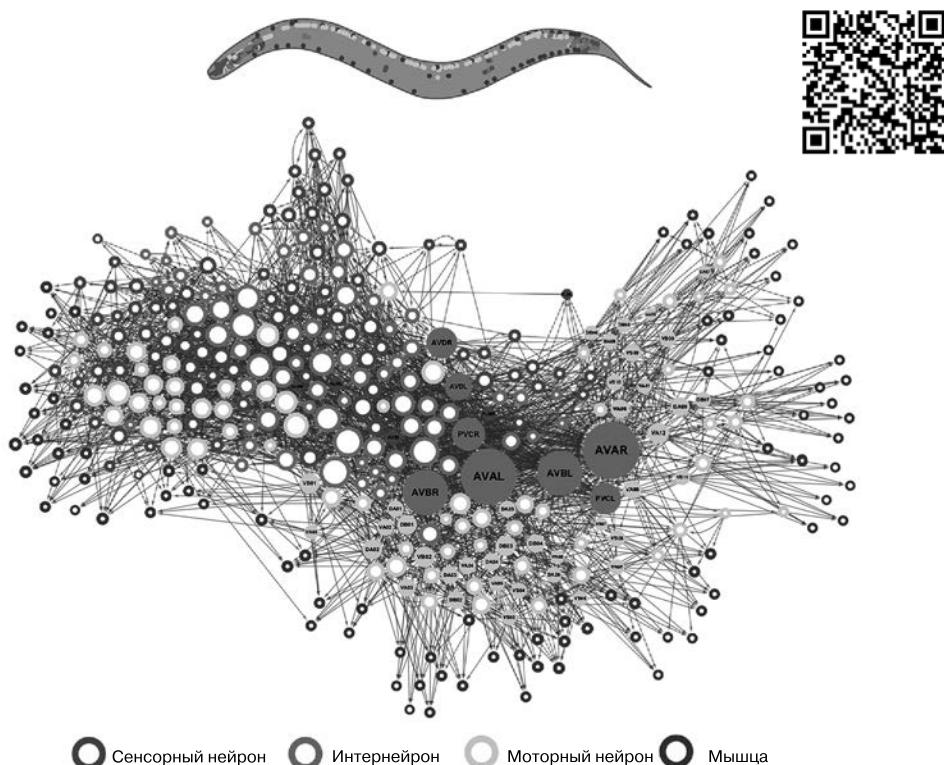
Существует фундаментальное отличие простого геометрического преобразования входных данных в выходные, которое выполняют модели глубокого обучения, от того, как думают и обучаются люди. Дело не только в том, что люди учатся на своем опыте, а не на явных обучающих примерах. Человеческий мозг работает совершенно иначе, чем дифференцируемая параметрическая функция.

Давайте подойдем к этой теме с другой стороны и спросим себя: зачем по-надобился интеллект? Почему он возник? Мы можем только предполагать, но эти предположения будут достаточно обоснованными. Мы можем начать с изучения мозга: органа — вместилища интеллекта. Мозг — эволюционная адаптация, механизм, развивавшийся в течение сотен миллионов лет путем случайных проб и ошибок в ходе естественного отбора, — существенно расширил способность организмов адаптироваться к окружающей среде. Мозг появился более полутора миллиарда лет назад как *хранилище и исполнительный механизм поведенческих программ*. «Поведенческие программы» — это всего лишь наборы инструкций, которые заставляют организм реагировать на окружающую среду: «Если происходит это, делай то». Они связывают сенсорные входы организма с центрами управления двигательной системой. Вначале мозг жестко кодировал поведенческие программы в виде наборов нейронных связей, которые позволяли организму адекватно реагировать на поступающие сенсорные сигналы. Так до сих пор работает мозг насекомых — мух, муравьев, нематод (рис. 14.4) и т. д. Поскольку первоначальным «исходным кодом» этих программ была ДНК, которая расшифровывалась как набор нейронных связей, эволюция внезапно получила возможность осуществлять  *поиск в поведенческом пространстве* практически неограниченным количеством способов — это серьезный эволюционный сдвиг.

Эволюция была программистом, а мозг — компьютером, скрупулезно выполняющим заданный эволюцией код. Поскольку нейронные связи являются весьма распространенной вычислительной основой, сенсомоторное пространство всех видов, обладающих мозгом, может резко начать расширяться. Глаза, уши,

нижние челюсти, четыре ноги, 24 ноги — если у вас есть мозг, эволюция любезно разработает уникальную поведенческую программу, которая эффективно задействует все, что у вас есть. Мозг может справиться с любой модальностью или комбинацией модальностей, которую вы ему подбросите.

Имейте в виду, что на ранних этапах эволюции мозг не был носителем разума. Скорее, он был *автоматом*, просто выполняющим поведенческие программы, которые были жестко закодированы в ДНК организма. Их можно назвать в том же смысле интеллектуальными, в каком «интеллектуальным» является термостат. Или программа сортировки списка. Или... обученная глубокая нейронная сеть, искусственно созданная. Это важное замечание, на которое стоит обратить внимание: так в чем же разница между автоматами и настоящими носителями интеллекта?



**Рис. 14.4.** Модель мозга нематоды: поведенческий автомат, запрограммированный естественной эволюцией. Рисунок создан Эммой Тоулсон (взят из статьи Яна с соавторами *Network control principles predict neuron function in the *Caenorhabditis elegans* connectome*, опубликованной в журнале *Nature*, октябрь 2017 года)

### 14.2.3. Локальное и экстремальное обобщение

Французский философ и ученый XVII века Рене Декарт в 1637 году написал поучительный комментарий, который прекрасно отражает это отличие, — задолго до появления ИИ и первого механического компьютера (который его коллега Паскаль создал пятью годами позже). Декарт говорит об автоматах:

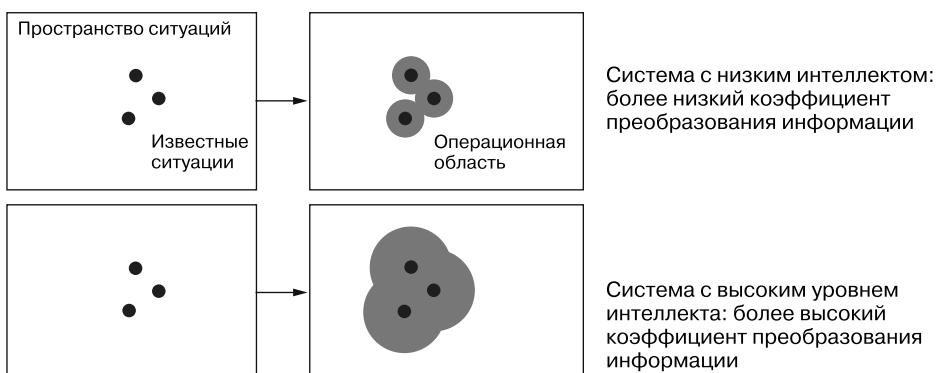
«Хотя такая машина многое могла бы сделать так же хорошо и, возможно, лучше, чем мы, в другом она непременно оказалась бы несостоительной, и обнаружилось бы, что она действует не сознательно, а лишь благодаря расположению своих органов».

*Рене Декарт, Рассуждения о методе, 1637 год.*

Вот оно. Интеллект характеризуется *пониманием*, а понимание подтверждается *обобщением* — способностью справиться с любой новой ситуацией. Как отличить студента, зазубрившего ответы на экзаменационные вопросы последних трех лет, но ничего в них не понимающего, и студента, который действительно проработал материал? Дайте им совершенно новую задачу. Автомат статичен, он может выполнять определенные действия в определенном контексте («если это, то это»), тогда как носитель интеллекта способен на лету адаптироваться к новым, неожиданным ситуациям. Когда автомат подвергается воздействию чего-то, что не соответствует его «программе» (что бы ни подразумевалось под этим термином — программа, написанная человеком или созданная эволюцией, или неявная программа модели, полученная в процессе ее подгонки под обучающий набор данных), он окажется не в состоянии предпринять адекватные действия. Между тем носители интеллекта будут использовать имеющиеся знания, чтобы найти путь вперед.

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. Эта способность является ключевой отличительной чертой интеллекта, которым обладают люди и высшие животные.

Это резко контрастирует с действиями глубоких сетей, которые я называю *локальным обобщением*. Примитивные автоматы вообще не обладают способностью к обобщению — они неспособны справиться с задачами, на выполнение которых их не запрограммировали заранее. Словарь в языке Python или простая программа — сборник ответов на вопросы, реализованная как жестко определенная последовательность операторов if-then-else, — попадают в эту категорию. Глубокие сети действуют чуть лучше: они могут успешно обрабатывать входные данные, немного отличающиеся от того, с чем они знакомы, — именно это делает их полезными. Наша модель классификации изображений кошек и собак из главы 8 может отличать кошек от собак, изображения которых она раньше не видела, но только если эти изображения достаточно близки к тем, на которых она обучалась. Однако глубокие сети способны только на *локальное обобщение* (рис. 14.5): 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. Глубокие сети могут обобщать только *известные неизвестные* — факторы вариации, которые ожидались во время разработки модели и широко представлены в обучающих данных: разные ракурсы камеры или условия освещения при фотографировании домашних животных. Это связано с тем, что глубокие сети реализуют обобщение посредством интерполяции на многообразии (как рассказывалось в главе 5): любой фактор вариации в пространстве входных данных должен иметь отражение в многообразии, которое они изучают. Вот почему обогащение базовых данных так полезно для улучшения способности глубокой сети к обобщению. В отличие от людей эти модели неспособны импровизировать в ситуациях, о которых известно мало или ничего не известно вовсе (например выигрыш в лотерею), которые имеют лишь абстрактные общие черты с прошлыми ситуациями.



**Рис. 14.5.** Локальное и экстремальное обобщение

Рассмотрим, например, задачу определения параметров запуска ракеты, которая должна сесть на Луну. Если для ее решения использовать глубокую сеть и обучить ее, применив 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 выработает соответствующее поведение уклонения. Однако при попадании в другой город сети придется забыть большую часть того, что она уже изучила, и перечувствовать заново. Люди, напротив, способны обучаться правилам безопасного поведения, не переживая фатального конца ни разу, — и снова благодаря своей способности абстрактного моделирования гипотетических ситуаций.

#### 14.2.4. Назначение интеллекта

Это различие между легко адаптируемыми носителями интеллекта и бездушными автоматами возвращает нас к эволюции мозга. Почему мозг — первоначально простой инструмент эволюции, предназначенный для развития поведенческих автоматов, — в конце концов обрел разум? Как и многое другое в эволюции, это произошло благодаря ограничениям естественного отбора.

Мозг отвечает за формирование поведения. Если бы набор ситуаций, с которыми приходится сталкиваться организму, был преимущественно статичен и известен заранее, то выбор модели поведения был бы простой задачей: эволюция просто определила бы правильный вариант путем проб и ошибок и жестко закодировала его в ДНК. Первая стадия эволюции мозга — мозг как автомат — уже была бы для этого оптимальной. Тем не менее сложность организма — а вместе с ним и сложность окружающей среды — продолжала возрастать, и ситуации, с которыми приходилось иметь дело животным, становились все более динамичными и непредсказуемыми. Посмотрите внимательно: любой день вашей жизни не похож ни на какой другой, прожитый вами или вашими эволюционными предками. Вы постоянно оказываетесь в новых, неожиданных ситуациях. Эволюция не может предугадать и сохранить для вас последовательность действий, которую вы должны выполнять, чтобы успешно пройти через очередной день. Эта последовательность постоянно должна генерироваться на лету.

В итоге мозг, будучи инструментом формирования поведенческих реакций, принял во внимание эту необходимость и оптимизировался для адаптации, а не

для приспособления к фиксированному набору ситуаций. Вероятно, данный сдвиг происходил много раз на протяжении всей эволюции, что приводило к появлению высокоорганизованных животных в очень отдаленных эволюционных ветвях — человекообразных обезьян, осьминогов, воронов и других. Интеллект — это ответ на неожиданности, возникающие в сложных, динамичных экосистемах.

Такова природа интеллекта: это замечательная способность эффективно использовать имеющуюся информацию для успешного выхода из сложных, постоянно меняющихся ситуаций. То, что Декарт называет пониманием, является ключом к ней: это способность применить свой прошлый опыт в выработке универсальных абстракций, которые можно быстро адаптировать под новое окружение и достигнуть экстремального обобщения.

### **14.2.5. Восхождение по спектру обобщения**

Историю эволюции биологического интеллекта в упрощенном виде можно представить как медленное движение вверх по *спектру обобщения*. Все началось с мозга, похожего на автомат, которому под силу было только локальное обобщение. Со временем эволюция привела к появлению организмов, способных к более широкому обобщению и существованию во все более сложных и изменчивых условиях. Наконец, в последние несколько миллионов лет — мгновение с точки зрения эволюции — у некоторых видов гоминидов выработалась склонность к развитию биологического интеллекта, способного к экстремальным обобщениям, что ускорило начало антропоцене и навсегда изменило историю жизни на Земле.

Развитие ИИ в последние 70 лет поразительно напоминает эволюцию. Первые системы ИИ были чистыми автоматами, как, например, программа виртуального собеседника ELIZA, написанная в 1960-х годах, или SHRDLU<sup>1</sup> — ИИ 1970-х, способный манипулировать простыми объектами, следя командам на естественном языке. В 1990-х и 2000-х годах появились системы машинного обучения, способные к локальному обобщению и успешно справляющиеся с некоторым уровнем неопределенности и новизны. В 2010-х годах способности глубокого обучения к локальному обобщению выросли еще больше и позволили инженерам использовать намного большие наборы данных и более выразительные модели.

В настоящее время мы можем оказаться на пороге следующего эволюционного шага. Растет интерес к системам, способным достичь широкого обобщения, которое я определяю как умение иметь дело с неизвестными неизвестными<sup>2</sup>

<sup>1</sup> Winograd T. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. 1971.

<sup>2</sup> Отсылка на цитату Дональда Рамсфельда: <https://ru.citaty.net/tsitaty/651068-donald-ramsfeld-est-izvestnye-izvestnye-veshchi-o-kotorykh-my-zn/>. — Примеч. пер.

в рамках одной широкой области задач (включая ситуации, с которыми система не обучалась справляться и которые ее создатели не могли предвидеть). Например, беспилотный автомобиль, способный безопасно преодолевать любые трудности, которые вы ему подбрасываете, или домашний робот, который может пройти «кофейный тест» Возняка — войти в случайную кухню и приготовить чашку кофе<sup>1</sup>. Развивая глубокое обучение и одновременно кропотливо создавая абстрактные модели мира вручную, мы уже видим заметный прогресс в достижении этих целей.

Однако на данный момент ИИ остается ограниченным *когнитивной автоматизацией*. Слово «интеллект» появилось в названии «искусственный интеллект» по ошибке — было бы правильнее назвать нашу сферу «искусственное познание», где «когнитивная автоматизация» и «искусственный интеллект» были бы двумя почти независимыми областями. При таком делении «искусственный интеллект» был бы новой областью, в которой почти все еще предстоит открыть.

Я не хочу преуменьшать достижения глубокого обучения. Когнитивная автоматизация несет невероятную практическую пользу, и способность моделей глубокого обучения автоматизировать задачи, основанные исключительно на данных, представляет собой особенно мощную ее форму, гораздо более практическую и универсальную, чем явное программирование. Тщательно подготовленные и детально проработанные модели могут совершить прорыв практически в любой отрасли. Но до человеческого (или животного) интеллекта им еще очень далеко. Пока что наши модели способны только на локальное обобщение: они отображают пространство X в пространство Y с помощью непрерывных геометрических преобразований, выведенных из плотной выборки соответствующих точек данных X и Y, и любое нарушение в пространствах X или Y нарушает это отображение. Они могут обобщать только такие новые ситуации, которые похожи на предыдущие, тогда как человеческое познание способно к экстремальным обобщениям, быстрой адаптации к радикально новой обстановке и планированию даже для далекого будущего.

### 14.3. КУРС НА УВЕЛИЧЕНИЕ УНИВЕРСАЛЬНОСТИ В ИИ

Чтобы снять некоторые обсужденные нами ограничения и создать ИИ, способный конкурировать с человеческим мозгом, нужно отойти от простого отображения ввода в вывод и перейти к *рассуждениям* и *абстракциям*. В следующих нескольких пунктах мы рассмотрим, как может выглядеть дальнейший путь.

---

<sup>1</sup> Fast Company. Wozniak: Could a Computer Make a Cup of Coffee? (Март 2010), <http://mng.bz/pJMP>.

### **14.3.1. О важности постановки верной цели: правило выбора кратчайшего пути**

Биологический интеллект был ответом на заданный природой вопрос. По аналогии, чтобы получить правильный искусственный интеллект, мы должны научиться задавать правильные вопросы.

Эффект, который вы постоянно наблюдаете при проектировании систем, является результатом действия *правила выбора кратчайшего пути*: сосредоточившись на оптимизации одной метрики успеха, вы достигаете поставленной цели, но за счет всего остального, что не было этой метрикой охвачено. В конечном итоге вы действуете все доступные короткие пути к цели, а ваши творения в результате формируются стимулами, которые вы сами же и задаете.

Подобное часто можно наблюдать на соревнованиях по машинному обучению. В 2009 году компания Netflix провела конкурс, в котором команде, набравшей наибольшее количество баллов за систему предсказания зрительских предпочтений, был обещан приз в размере 1 миллиона долларов. В итоге они так и не начали использовать систему, созданную командой-победителем, потому что она была слишком сложной и требовательной к вычислениям. Победители оптимизировали систему для достижения максимальной точности предсказания и таким образом реализовали поставленную перед ними цель, однако сделали это за счет всех остальных желательных характеристик: стоимости логического вывода, простоты поддержки и объяснимости. Правило выбора кратчайшего пути справедливо и для большинства соревнований на Kaggle: модели, созданные победителями данных конкурсов, за редким исключением, практически непригодны для использования в производстве.

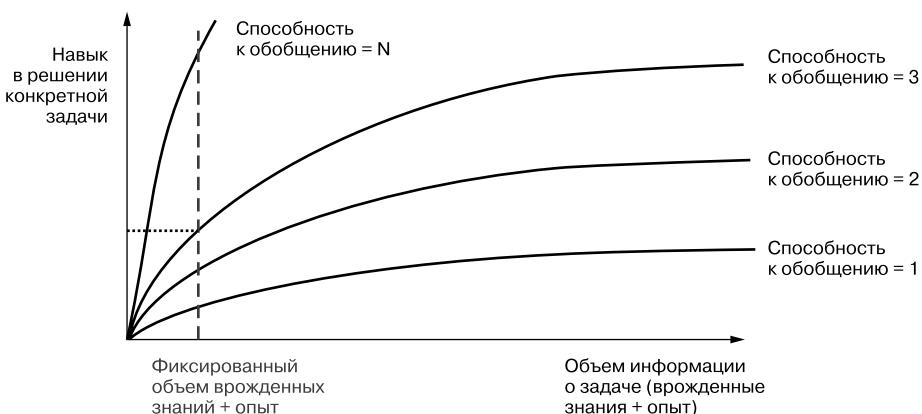
Правило выбора кратчайшего пути в последние несколько десятилетий реализовывалось в ИИ повсюду. В 1970-х годах психолог и пионер информатики Аллен Ньюэлл, обеспокоенный отсутствием значимого прогресса в развитии правильной теории познания в данной области, предложил новую великую цель для ИИ: играть в шахматы. Он обосновал это тем, что шахматы требуют от людей способности воспринимать, рассуждать и анализировать, запоминать, изучать источники и т. д. Аллен считал, что машина, которая будет играть в шахматы, тоже должна обладать этими качествами. Ведь так?

Более двух десятилетий спустя его мечта сбылась: в 1997 году компьютер Deep Blue, построенный компанией IBM, обыграл Гарри Каспарова — лучшего шахматиста в мире. Однако исследователям пришлось признать, что создание ИИ — чемпиона по шахматам практически не продвинуло их в познании человеческого интеллекта. Алгоритм Alpha-Beta, заложенный в Deep Blue, не был моделью человеческого мозга и не мог использоваться для других задач, кроме участия в подобных играх. Оказалось, что спроектировать ИИ, который мог бы

играть только в шахматы, намного проще, чем создать искусственный разум, так что исследователи снова пошли по кратчайшему пути.

До сих пор основной мерой успеха в области ИИ было решение конкретных задач: от шахмат до го, от классификации рукописных цифр из набора MNIST до сортировки изображений в наборе ImageNet, от аркадных игр Atari до StarCraft и DotA 2. В результате в истории развития области появилось множество «успехов», когда мы на самом деле придумывали, как решать задачи *без участия интеллекта*.

Если это утверждение показалось вам неожиданным, имейте в виду, что человеческий интеллект не характеризуется способностями в какой-либо конкретной задаче — скорее, это умение адаптироваться к новизне, приобретать новые навыки и осваивать решение не встречавшихся ранее проблем. Фиксируя задачу, вы делаете возможным сколь угодно точное описание того, что необходимо получить, — либо путем прямого кодирования предоставленных человеком знаний, либо путем передачи огромных объемов данных. Вы даете инженерам возможность «купить» больше навыков для своего ИИ, просто добавляя данные или программируя знания, не увеличивая при этом способность ИИ к обобщению (рис. 14.6). Если у вас есть почти бесконечный набор обучающих данных, то даже очень грубый алгоритм, такой как поиск ближайшего соседа, сможет проявлять сверхчеловеческие способности, играя в видеоигры. Это же справедливо к почти бесконечному количеству написанных человеком операторов if-then-else. Но стоит внести небольшое изменение в правила игры — такое, к которому человек может мгновенно приспособиться, — и вам придется повторно обучить или перенастроить неинтеллектуальную систему.



**Рис. 14.6.** Система с низкой способностью к обобщению может выработать навык решения фиксированной задачи сколь угодно хорошо при наличии неограниченного объема информации о ней

Иначе говоря, фиксируя задачу, вы устраниете необходимость осваивать неопределенность и новизну, а поскольку природа интеллекта заключается как раз в способности справляться с неопределенностью и новизной, вы фактически устраниете потребность в интеллекте. Найти неинтеллектуальное решение конкретной задачи всегда проще, чем решить общую проблему интеллекта, поэтому данный кратчайший путь вы выберете в 100 % случаев. Люди могут использовать свой интеллект для приобретения навыков в любой новой задаче, но обратный путь — от набора специальных навыков к общему интеллекту — невозможен.

### **14.3.2. Новая цель**

Чтобы сделать искусственный интеллект разумным и наделить его способностью справляться с невероятной изменчивостью реального мира, сначала нужно отойти от стремления к оттачиванию *навыков, специфичных для конкретной задачи*, и нацелиться на способность к обобщению. Нам нужны новые метрики для оценки прогресса, которые помогут разрабатывать все более интеллектуальные системы. Метрики, которые укажут правильное направление и дадут четкий сигнал обратной связи. Пока мы ставим перед собой цель «создать модель, решающую задачу X», правило выбора кратчайшего пути будет рабочим — в итоге мы получим модель, решающую задачу X, и точка.

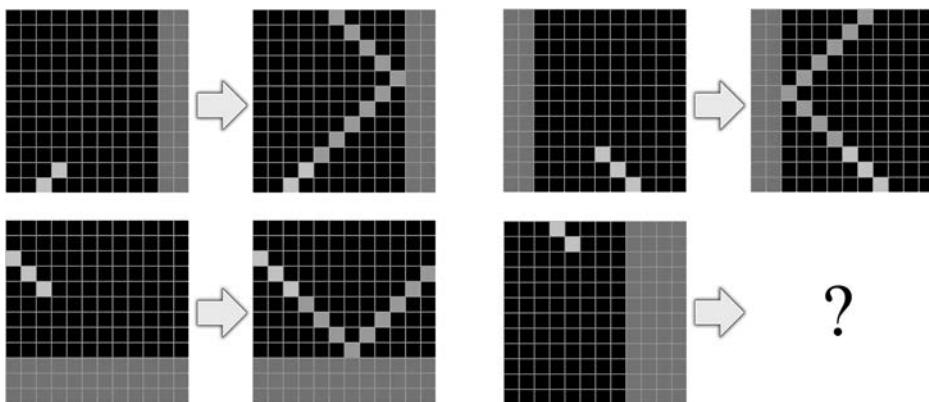
На мой взгляд, интеллект можно точно определить как *коэффициент эффективности*: отношение между доступным *объемом значимой информации* о мире (которая может быть суммой прошлого опыта и приобретенных знаний) и широтой *будущей области принятия верных решений* — набором новых ситуаций, в которых вы сможете действовать соответствующим образом (рассматривайте его как свой *набор навыков*). Более интеллектуальный агент сможет справиться с более широким набором будущих задач и ситуаций, используя меньший объем прошлого опыта. Чтобы измерить эту зависимость, достаточно зафиксировать информацию, доступную вашей системе, — ее опыт и приобретенные знания — и оценить ее качество на наборе эталонных ситуаций или задач, существенно отличных от тех, что система видела во время обучения. Попытка максимизировать данное отношение должна привести вас к разуму. Крайне важно избегать жульничества: тестировать систему нужно только на задачах, на решение которых она не была запрограммирована или обучена, — на задачах, которые *создатели системы не могли предвидеть*.

В 2018 и 2019 годах я разработал эталонный набор данных под названием «Корпус абстракций и рассуждений» (Abstraction and Reasoning Corpus, ARC)<sup>1</sup>, призванный визуализировать определение интеллекта. Предполагается, что он доступен и машинам, и людям и очень похож на такие тесты для определения уровня IQ,

---

<sup>1</sup> Chollet F. On the Measure of Intelligence. 2019, <https://arxiv.org/abs/1911.01547>.

как прогрессивные матрицы Равена. Во время тестирования вы увидите ряд задач. Каждая задача объясняется с помощью трех или четырех примеров, принимающих форму начальной и соответствующей ей конечной сетки (рис. 14.7). Затем для допуска к следующему заданию вам будут предложены новая начальная сетка и три попытки на создание правильной конечной сетки.



**Рис. 14.7.** Задача ARC. Характер задачи демонстрируется парой примеров начальной и конечной сетки. После чего для новой начальной сетки вы должны построить соответствующую конечную сетку

От тестов IQ корпус ARC отличают две уникальных черты. Во-первых, ARC стремится оценить способность к обобщению, проверяя вас только на задачах, с которыми вы прежде не сталкивались. Следовательно, ARC — это *игра, которая не дает возможности потренироваться*, по крайней мере теоретически: задачи, на которых выполняется тестирование, имеют свою уникальную логику, которую вам придется осваивать по ходу дела. Вы не сможете просто запомнить конкретные стратегии из прошлых задач.

Кроме того, корпус ARC пытается учитывать *врожденные знания*, которые у вас уже есть. Вы никогда не подходите к новой задаче с нуля — вы используете собственные навыки и опыт. ARC исходит из того, что все испытуемые имеют так называемые «априорные знания» из «системы знаний», с которой люди рождаются. В отличие от теста на IQ задачи ARC никогда не вовлекают приобретенные знания, такие как, например, знание какого-либо языка.

Неудивительно, что методы, основанные на глубоком обучении (включая модели, обученные на очень больших объемах внешних данных, такие как GPT-3), оказались совершенно неспособными справиться с тестом ARC: эти задачи не решаются методом интерполяции, следовательно, они плохо подходят для подгонки моделей. Между тем обычные люди, сталкиваясь с ними,

не испытывают никаких проблем и решают задачи с первого раза без всякой практики. Подобные ситуации, когда пятилетние дети способны естественно и без затруднений выполнять что-то, что абсолютно невозможно для современных технологий искусственного интеллекта, служат четким сигналом, что происходит что-то непонятное — что-то, что мы упускаем.

Что нужно для решения задач из корпуса ARC? Надеюсь, данный вопрос заставит вас задуматься. В этом весь смысл ARC: дать цель другого рода, которая подтолкнет вас в новом направлении — полагаю, в продуктивном. А теперь давайте кратко рассмотрим ключевые ингредиенты, которые вам понадобятся, если вы решитесь ответить на этот вопрос.

## **14.4. РЕАЛИЗАЦИЯ ИНТЕЛЛЕКТА: НЕДОСТАЮЩИЕ ИНГРЕДИЕНТЫ**

Итак, вы узнали, что интеллект — это нечто гораздо большее, чем интерполяция в скрытом многообразии, которую выполняет глубокое обучение. А что нужно, чтобы создать настоящий интеллект? Какие основные компоненты ускользают от нас в настоящее время?

### **14.4.1. Интеллект как чувствительность к абстрактным аналогиям**

Интеллект — это способность использовать прошлый опыт (и накопленные знания) для ориентации в новых, неожиданных условиях. Однако если бы будущие ситуации, в которых вам пришлось бы оказаться, были *по-настоящему новыми* — не имеющими ничего общего с теми, с которыми вы сталкивались ранее, — то вы не смогли бы отреагировать на них, какими бы умными вы ни были.

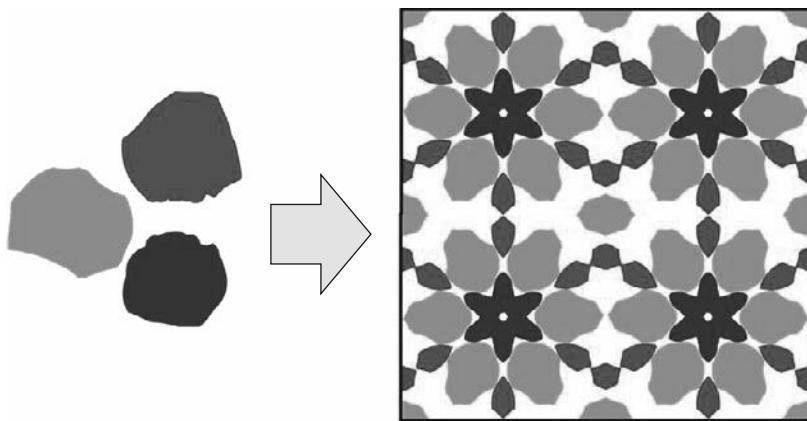
Интеллект помогает справиться с ситуацией, потому что в действительности не случается ничего беспрецедентного. Столкнувшись с чем-то новым, мы можем разобраться в нем, проводя аналогии с прошлым опытом, формулируя объяснения в терминах абстрактных понятий, накопленных за прошедшее время. Человек из XVII века, впервые увидевший реактивный самолет, мог бы описать его как большую металлическую птицу, которая не машет крыльями. Автомобиль? Это повозка без лошади. Преподаватель физики в школе может рассказать детям, что электрический ток подобен течению воды в трубе или что пространство-время похоже на резиновый лист, форма которого искажается тяжелыми предметами.

Помимо таких четких, явных аналогий мы постоянно проводим более мелкие, неявные — каждую секунду, в каждой мысли. Аналогии помогают нам

ориентироваться в жизни. Зашли в новый супермаркет? Имея опыт походов в подобные магазины, вы в нем не потеряетесь. Говорите с кем-то незнакомым? Он может напомнить вам кого-то, кого вы встречали раньше. Даже кажущиеся случайными узоры вроде формы облаков вызывают в нашем сознании яркие образы — слона, корабля, рыбы.

Эти аналогии существуют не только в нашем сознании: сама физическая реальность полна изоморфизмов. Электромагнетизм подобен гравитации. Все животные по структуре организма похожи друг на друга, что обусловлено общим происхождением. Кристаллы кварца похожи на кристаллы льда. И так далее.

Я называю это *гипотезой калейдоскопа*: кажется, что наше восприятие мира отличается невероятной сложностью и нескончаемой новизной, но все в этом море сложности похоже на все остальное. Число *的独特ных атомов смысла*, необходимое для описания вселенной, в которой вы живете, относительно невелико, и все окружающее вас является некоторой рекомбинацией этих атомов. Несколько базовых компонентов и бесконечные вариации — очень похоже на то, что происходит внутри калейдоскопа, где несколько осколков цветного стекла отражаются системой зеркал, создавая богатые, казалось бы, постоянно меняющиеся узоры (рис. 14.8).



**Рис. 14.8.** Калейдоскоп создает богатое (но повторяющееся) многообразие узоров всего из нескольких осколков цветного стекла

Способность к обобщению (интеллект) — это умение анализировать свой опыт, идентифицировать атомы смысла, которые можно повторно применять во многих ситуациях. После извлечения они превращаются в *абстракции*. Всякий раз, сталкиваясь с чем-то новым, вы осмысливаете его, пропуская через накопленный набор абстракций. Как вы идентифицируете повторное использование атомов смысла? Просто замечаете сходство — аналогии. Если что-то повторяется дважды, то оба экземпляра должны иметь одно начало, как в калейдоскопе.

Абстракция — это двигатель интеллекта, а построение аналогий — это двигатель, производящий абстракции.

Проще говоря, интеллект — это чувствительность к абстрактным аналогиям, и ничего больше. Если у вас высокая чувствительность к аналогиям, вы будете извлекать мощные абстракции из небольшого опыта и сможете использовать эти абстракции для ориентации в обширном пространстве будущего опыта. Вы сможете максимально эффективно преобразовывать прошлый опыт в способность справляться с новизной в будущем.

### **14.4.2. Два полюса абстракции**

Если интеллект — это чувствительность к аналогиям, то разработка искусственного интеллекта должна начинаться с создания пошагового алгоритма проведения аналогий. Проведение аналогии начинается со *сравнения вещей друг с другом*. Важно отметить, что есть два способа сравнения, из которых возникает два разных вида абстракций — два способа мышления, подходящих для решения своего круга задач. Вместе эти два полюса абстракции образуют основу нашего мышления.

Первый способ основан на сопоставлении вещей друг с другом — это *сравнение по сходству*, порождающее *ценностно-центрические аналогии*. Второй способ — это *точное структурное соответствие*, порождающее *программно-центрические аналогии* (или структурно-центрические аналогии). В обоих случаях все начинается с исследования экземпляров чего-то и их сопоставления, в ходе которого рождается *абстракция*, определяющая общие элементы базовых экземпляров. Способы различаются только тем, как определяется сходство экземпляров и как на их основе формируется абстракция. Рассмотрим внимательно каждый способ.

#### **Ценностно-центрическая аналогия**

Представьте, что у себя на заднем дворе вы нашли несколько жучков разных видов. Рассмотрев их, вы отметили наличие определенного сходства между ними. Одни были более похожи друг на друга, другие — менее: понятие сходства неявно выражается гладкой, непрерывной *функцией расстояния*, определяющей скрытое многообразие, в котором живут ваши экземпляры. Насмотревшись на достаточное количество жучков, вы можете начать группировать похожие экземпляры и объединять их в набор *прототипов*, отражающих общие внешние особенности каждого набора (рис. 14.9). Прототипы являются абстрактными: они не похожи ни на один конкретный экземпляр, который вы видели, но обладают свойствами, общими для всех экземпляров. Увидев нового жучка, вам не нужно сравнивать его с каждым жучком, которого вы видели раньше, чтобы определить, к какой группе его отнести. Вы можете просто сравнить его с несколькими прототипами, выбрать ближайший — *категорию* жучка — и использовать его для полезных прогнозов: может ли жук вас укусить? Будет ли он есть ваши яблоки?



**Рис. 14.9.** Ценностно-центрическая аналогия связывает экземпляры через понятие сходства и помогает получить абстрактные прототипы

Звучит знакомо? Данная схема довольно точно описывает работу машинного обучения без учителя (например, алгоритм кластеризации  $K$  средних). В общем случае все виды современного машинного обучения, с учителем или без, работают путем исследования скрытых многообразий, описывающих пространство экземпляров, закодированных с помощью прототипов. (Вспомните визуализацию признаков, определяемых сверточными сетями, в главе 9. Это были визуальные прототипы.) Ценностно-центрическая аналогия — это разновидность аналогии, позволяющая моделям глубокого обучения выполнять локальное обобщение.

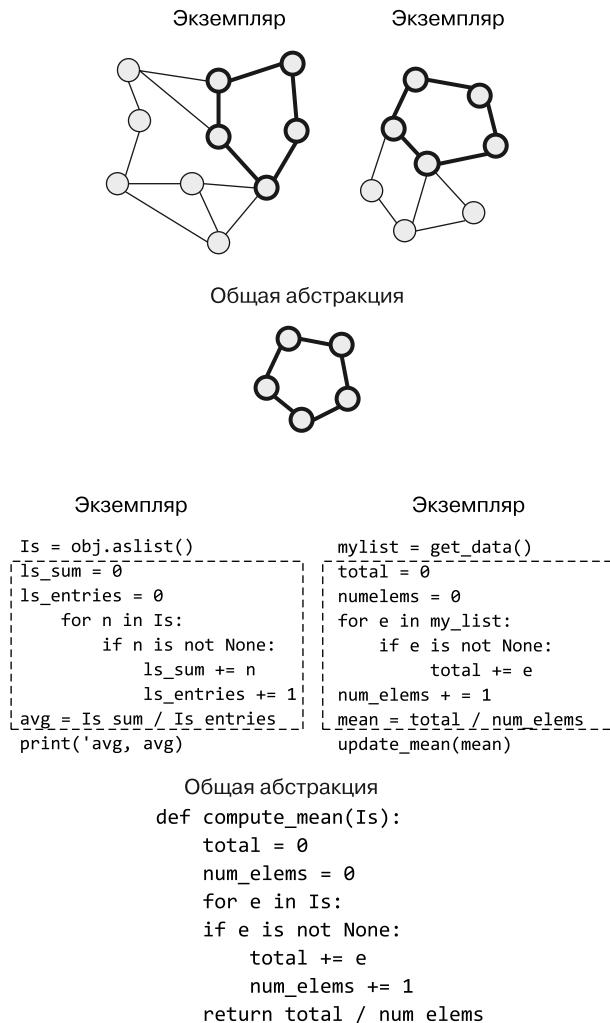
На ней также основаны многие наши когнитивные способности. Будучи человеком, вы постоянно проводите ценностно-центрические аналогии. Данный тип абстракции лежит в основе *распознавания образов, восприятия и интуиции*. Способность решать задачи не задумываясь в значительной степени основана на ценностно-центрической аналогии. Если вы смотрите фильм и начинаете подсознательно классифицировать персонажей по типам, то это ценностно-центрическая абстракция.

### Программно-центрическая аналогия

Важно отметить, что познание — это нечто большее, чем немедленная, приближенная, интуитивная классификация, основанная на ценностно-центрической аналогии. Есть еще один тип механизмов генерации абстракций, более медленный, точный и осмысленный: программно-центрическая (или структурно-центрическая) аналогия.

Разрабатывая программное обеспечение, вы часто пишете разные функции или классы, которые, кажется, имеют много общего. Заметив эту избыточность, вы начинаете задаваться вопросом: «Можно ли написать более абстрактную функцию, выполняющую ту же работу, которую можно было бы использовать

дважды? Можно ли определить абстрактный базовый класс и унаследовать его в обоих моих классах?» Такое определение абстракции соответствует программно-центрической аналогии. Вы не пытаетесь сравнить свои классы и функции по *внешнему сходству*, подобно тому как сравниваете два человеческих лица с помощью неявной функции расстояния. Вас больше интересуют части, имеющие *одинаковую структуру*. Вы ищете то, что называется *изоморфизмом подграфов* (рис. 14.10): программы могут быть представлены в виде графов операторов, и вы пытаетесь найти подграфы (подмножества операторов), общие для разных программных компонентов.



**Рис. 14.10.** Программно-центрическая аналогия идентифицирует и изолирует изоморфные подструктуры в разных экземплярах

Данный способ проведения аналогий посредством точного структурного сопоставления различных дискретных структур вовсе не является исключительным для специализированных областей, таких как информатика или математика, — вы постоянно применяете его, сами того не замечая. Он лежит в основе *рассуждений, планирования* и общей концепции *точности* (в отличие от интуиции). Всякий раз, думая об объектах, связанных друг с другом дискретной сетью отношений (а не непрерывной функцией сходства), вы используете программно-ориентированные аналогии.

### **Познание как соединение обоих видов абстракции**

Сравним эти два полюса абстракции (табл. 14.1).

**Таблица 14.1.** Два полюса абстракции

Ценностно-центрическая абстракция	Программно-центрическая абстракция
Связь экземпляров определяется расстоянием	Связь экземпляров определяется структурным сходством
Непрерывная, основана на геометрии	Дискретная, основана на топологии
Создает абстракции «усреднением» экземпляров в «прототипы»	Создает абстракции выделением из экземпляров изоморфных подструктур
Лежит в основе восприятия и интуиции	Лежит в основе рассуждений и планирования
Немедленная, приблизительная, интуитивная	Медленная, точная, строгая
Требуется большой опыт для получения надежных результатов	Эффективна с точки зрения опыта, может основываться всего на двух экземплярах

Все наши действия и мысли являются собой комбинацию этих двух типов абстракций. Вам придется потрудиться, чтобы отыскать задачу, для решения которой достаточно только какого-то одного из них. Даже такая, казалось бы, задача «чистого восприятия», как распознавание объектов на фотографии, требует некоторого количества неявных рассуждений об отношениях между объектами, на которые вы смотрите. И даже такая, казалось бы, задача «чистого рассуждения», как поиск доказательства математической теоремы, требует значительной доли интуиции. Когда математик касается пером бумаги, у него уже есть смутное представление о направлении, в котором следует двигаться. Дискретные логические шаги, которые он предпринимает для достижения цели, направляются интуицией высокого уровня.

Эти два полюса дополняют друг друга, и именно их чередование делает возможным экстремальное обобщение. Никакой разум не будет полным без них обоих.

### 14.4.3. Недостающая половина картины

К настоящему моменту у вас должно появиться понимание, чего не хватает современному глубокому обучению: оно очень хорошо кодирует ценностно-центрическую абстракцию, но практически не способно генерировать программно-центрическую абстракцию. Тем временем человеческий интеллект — это тесное сочетание абстракций обоих типов, поэтому мы буквально упускаем половину того, что нам нужно (возможно, самую важную половину).

А теперь хочу кое-что уточнить. До сих пор я представлял каждый тип абстракции как отдельный от другого и даже противоположный. Но на практике эта пара больше похожа на спектр: вы можете рассуждать, встраивая дискретные программы в непрерывные многообразия, подобно тому как можно подогнать полиномиальную функцию под любой набор дискретных точек при наличии достаточноного количества коэффициентов. И наоборот, вы можете использовать дискретные программы для эмуляции функций непрерывного расстояния — в конце концов, применяя методы линейной алгебры для вычислений на компьютере, вы работаете с непрерывными пространствами, используя исключительно дискретные программы, оперирующие единицами и нулями.

Однако есть и такие задачи, которые лучше подходят для того или другого типа абстракций. Например, попробуйте обучить модель глубокого обучения сортировке списка из пяти чисел. Не могу сказать, что это невозможно, но на пути к цели вас ждет много разочарований. Чтобы добиться успеха, вам понадобится огромное количество обучающих данных, но даже в этом случае модель будет периодически ошибаться, получив новые числа. А если нужно отсортировать десять чисел, то для этого придется заново обучить модель на еще большем количестве данных. Между тем реализация алгоритма сортировки на Python занимает всего несколько строк — и получившаяся программа, однажды проверенная на паре примеров, будет безошибочно сортировать списки любого размера. Это довольно мощное обобщение: перейти от пары демонстрационных и тестовых примеров к программе, которая с успехом обрабатывает буквально любой список чисел.

С другой стороны, задачи восприятия с большим трудом решаются с применением дискретных процессов рассуждения. Попробуйте написать программу на чистом Python для классификации цифр MNIST без использования методов машинного обучения: вас ждет захватывающее приключение. Вы обнаружите, что кропотливо программируете функции, которые могут определить количество замкнутых циклов в цифре, координаты центра масс цифры и т. д. Написав тысячи строк кода, вы можете достичь... точности 90 % на контрольных данных. В задачах такого вида намного проще подобрать (обучить) параметрическую модель: она способна лучше работать с большим объемом доступных данных

и достигать гораздо более надежных результатов. Если у вас много данных и перед вами стоит задача, где применяется гипотеза многообразий, используйте глубокое обучение.

По этой причине маловероятно, что мы увидим развитие подхода, который сводил бы задачу рассуждения к множественной интерполяции или задачу восприятия — к дискретным рассуждениям. Для создания искусственного интеллекта необходимо разработать единый фреймворк, включающий поддержку обоих типов абстрактных аналогий. Давайте посмотрим, как это может выглядеть.

## 14.5. БУДУЩЕЕ ГЛУБОКОГО ОБУЧЕНИЯ

Зная, как действуют глубокие сети, понимая их ограничения и недостатки, 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.

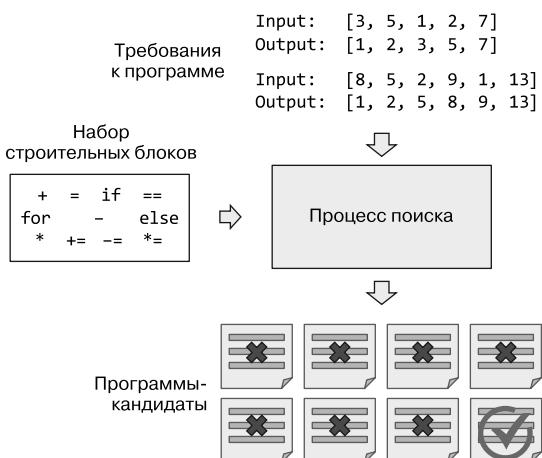
- *Модели, более близкие к универсальным компьютерным программам*, построенные на основе более широкого набора примитивов, чем современные дифференцируемые слои. Именно так мы приблизимся к возможности моделирования рассуждений и обобщения, отсутствие которой является основным недостатком современных моделей.
- *Слияние глубокого обучения и дискретного поиска в программных пространствах*, причем первое обеспечивает возможности восприятия и интуиции, а второе — возможности рассуждения и планирования.
- *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 неважно, 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.

### 14.5.1. 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.

Ситуация вскоре может измениться благодаря *синтезу программ* — довольно узкой области в настоящее время, но, думаю, в ближайшие десятилетия она будет расти и развиваться. Синтез программ заключается в создании простых программ с использованием алгоритма поиска (возможно, генетического поиска, как в *генетическом программировании*) для исследования обширного пространства возможных программ (рис. 14.11). 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 (табл. 14.2).



**Рис. 14.11.** Схематическое представление синтеза программы: опираясь на требования к программе и набор строительных блоков, механизм поиска собирает строительные блоки в программы-кандидаты, которые затем проверяются на соответствие требованиям. Поиск продолжается до тех пор, пока не будет найдена подходящая программа

**Таблица 14.2.** Машинное обучение и синтез программ

Машинное обучение	Синтез программ
Модель: дифференцируемая параметрическая функция	Модель: граф операторов из языка программирования
Механизм: градиентный спуск	Механизм: дискретный поиск (например, генетический поиск)
Чтобы получить надежный результат, необходим большой объем данных	Не требует много данных, может работать, имея пару обучающих примеров

Синтез программ — это одно из возможных средств поддержки программно-центрической абстракции в наши системы ИИ. Недостающая часть мозаики. Ранее я упоминал, что методы глубокого обучения совершенно непригодны для решения задач из корпуса ARC — теста для проверки интеллектуальности, ориентированной на рассуждения. Между тем даже первые робкие подходы к синтезу программ на этом teste дают многообещающие результаты.

### 14.5.2. Сочетание глубокого обучения и синтеза программ

Конечно, глубокое обучение никуда не денется. Синтез программ является не заменой, а дополнением. Это — второе полушарие, которого до сих пор не было в нашем искусственном мозге, а нам нужны оба полушария. Возможны два варианта их совместного использования.

1. Разработка систем, объединяющих модули глубокого обучения и дискретные алгоритмические модули.
2. Использование глубокого обучения для увеличения эффективности процесса поиска программы.

Давайте рассмотрим каждый из этих путей.

#### Интеграция модулей глубокого обучения и алгоритмических модулей в гибридные системы

Самые мощные современные системы ИИ являются гибридными: они используют и модели глубокого обучения, и созданные вручную программы манипулирования символами. В модели AlphaGo компании DeepMind, например, большая часть интеллекта спроектирована и запрограммирована опытными специалистами 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). Или возьмем автономные транспортные средства: беспилотный автомобиль способен справиться с широким спектром ситуаций, потому что поддерживает модель окружающего мира — буквальную трехмерную модель, полную допущений

и предположений, жестко закодированных инженерами. Эта модель постоянно обновляется с помощью модулей восприятия, основанных на глубоком обучении, которые связывают модель с окружением автомобиля.

Именно благодаря комбинации созданных человеком дискретных программ и непрерывно обучающихся моделей обеим системам – AlphaGo и беспилотным транспортным средствам – удалось достичь высочайшего уровня точности, что было бы невозможно при использовании любого из этих подходов в отдельности, например, для сквозной глубокой сети или программного обеспечения без элементов машинного обучения. Пока отдельные алгоритмические элементы таких гибридных систем кропотливо кодируются инженерами-людьми. Но в будущем подобные системы можно будет создавать без участия человека.

Как это будет выглядеть? 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` «защит» 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, что можно представить с помощью современных моделей глубокого обучения, и некоторые из этих программ могли бы достигать высочайшей степени обобщения. Важно отметить, что такие программы не будут дифференцируемыми от начала до конца (хотя некоторые модули сохранят свою дифференцируемость), поэтому их необходимо будет генерировать с помощью комбинации поиска дискретной программы и градиентного спуска.

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 (рис. 14.12). Вся система будет обучаться

без участия или с минимальным участием человека. Это должно существенно расширить круг задач, успешно решаемых с помощью машинного обучения — пространства программ, которые могут генерироваться автоматически с использованием соответствующих обучающих данных. Такие системы, как AlphaGo или даже RNN, можно рассматривать как доисторических предков подобных гибридных алгоритмически-геометрических моделей.



**Рис. 14.12.** Программа, сгенерированная одновременно на основе геометрических (распознавание шаблонов, предсказание) и алгоритмических (рассуждения, поиск, память) примитивов

### Использование глубокого обучения для направления поиска программ

В настоящее время синтез программ стоит перед серьезным препятствием: он чрезвычайно неэффективен. Он просто перебирает все возможные программы в пространстве поиска, пока не найдет ту, которая соответствует заданным требованиям. С усложнением требований или расширением словаря примитивов, используемых в программе, процесс поиска приводит к так называемому *комбинаторному взрыву*, когда число возможных программ, которые нужно оценить, растет даже быстрее, чем в экспоненциальной прогрессии. В результате сегодня синтез программ можно применять только для разработки очень коротких программ. В ближайшее время у вас не получится создать новую операционную систему для своего компьютера.

Чтобы двигаться вперед, нам нужно сделать синтез программ эффективным, приблизить его к тому, как люди пишут программы. Когда вы открываете редактор, чтобы написать сценарий, вы не думаете о каждой возможной программе, которую потенциально могли бы создать. Вы имеете в виду ограниченное число возможных подходов: вы можете использовать свое понимание задачи и свой прошлый опыт, чтобы резко сузить пространство возможных вариантов.

Глубокое обучение может помочь сделать то же самое для синтеза программ: каждую конкретную программу, которую необходимо сгенерировать, можно представить как принципиально дискретный объект, выполняющий неинтерполяционные манипуляции с данными, а имеющиеся данные — как *пространство*

*всех полезных программ*, очень похожее на непрерывное многообразие. Это означает, что модель глубокого обучения, обученная на результатах миллионов успешных попыток генерации программ, может выработать надежное *интуитивное представление о пути в пространстве программ*, по которому должен идти процесс поиска, чтобы перейти от требований к соответствующей программе, — точно так же инженер-программист может сразу интуитивно понять общую архитектуру сценария, который он собирается написать, и представить перечень промежуточных функций и классов, которые нужно использовать в качестве ступенек на пути к цели.

Помните, что человеческое мышление в значительной мере руководствуется ценностно-центрической абстракцией, то есть распознаванием образов и интуицией. То же относится и к синтезу. Я ожидаю, что подход на основе обобщенного управления поиском программы с помощью выявленной эвристики будет вызывать повышенный интерес исследователей в течение следующих 10–20 лет.

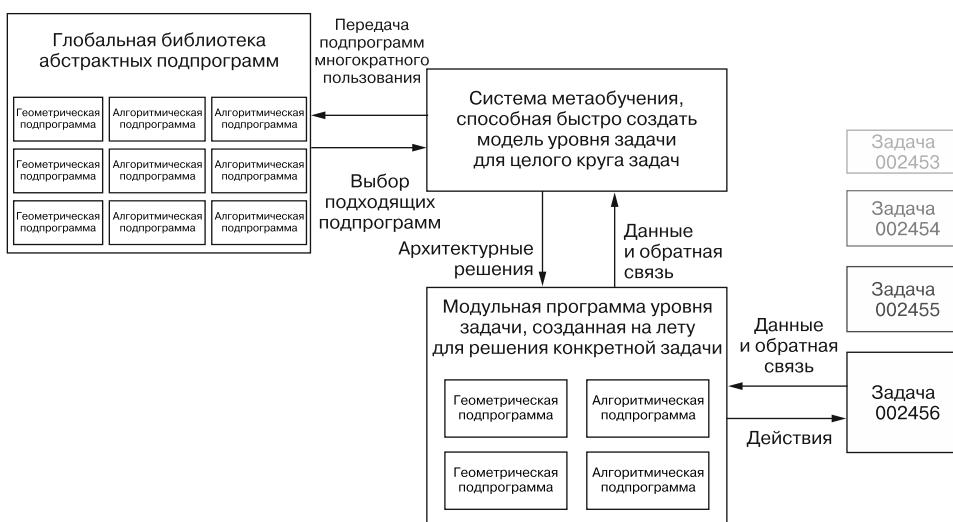
### **14.5.3. Непрерывное обучение и повторное использование модульных подпрограмм**

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 собою 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 9. В будущем я ожидаю, что в обиход 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 задачей 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 высокогоуровневых 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. 14.13). Такие подпрограммы могут быть геометрическими (модули глубокого обучения с предварительно выделенными представлениями) или алгоритмическими (ближе к библиотекам, которыми пользуются современные программисты).



**Рис. 14.13.** Система метаобучения, способная быстро разрабатывать модели для конкретных задач, используя примитивы многократного пользования (алгоритмические и геометрические), и таким способом достигать экстремального обобщения

#### 14.5.4. 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 *алгоритмические модули*, реализующие возможность формальных рассуждений, поиск и средства абстрагирования, с *геометрическими модулями*, обеспечивающими неформальное знание и распознавание шаблонов. Это позволит достичь сочетания ценностно-центрической и программно-центрической абстракций. AlphaGo или искусственный автопилот (системы, для создания которых потребовалось программное обеспечение, созданное вручную, и множество решений, принятых людьми) являются собой ранний пример того, как может выглядеть подобное сочетание символического 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 идентифицироваться системой метаобучения, превращаться в подпрограммы многократного пользования — подобно функциям и классам в разработке программного обеспечения — и добавляться в глобальную библиотеку.
- Процесс поиска возможных комбинаций подпрограмм для создания новых моделей будет дискретным процессом поиска (синтезом программы), но управляться он будет некоторой формой *интуиции в программном пространстве*, обеспечиваемой глубоким обучением.
- Упомянутая выше 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 в других видеоиграх, 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.

## **14.6. 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.

### **14.6.1. 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>). 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: с настройкой гиперпараметров, преодолением проблемы переобучения на проверочном наборе данных и ансамблированием моделей.

### 14.6.2. 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 *X* в данном случае подразумевается греческая буква «хи» —  $\chi$ ) — это открытый пре-принт-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 данной области. Это также поддерживает чрезвычайно высокий темп развития сферы: все новые находки немедленно становятся доступными для всех желающих.

Существенной проблемой является ежедневное появление в arXiv большого количества новых статей, поэтому невозможно 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. Однако уже есть некоторые инструменты, которые могут вам помочь: в частности, вы можете использовать Google Scholar (<https://scholar.google.com>), чтобы отслеживать выход новых публикаций определенных авторов.

### 14.6.3. Исследование экосистемы Keras

По состоянию на конец 2021 года там насчитывался примерно один миллион 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>. В частности, на странице <https://keras.io/guides> вы найдете обширный список руководств для разработчиков, а на странице <https://keras.io/examples> — десятки высококачественных примеров кода Keras. Обязательно ознакомьтесь с ними!
- Исходный код Keras можно найти по адресу <https://github.com/keras-team/keras>.
- Задавать вопросы, получать помощь и принимать участие в обсуждении проблем глубокого обучения можно, подписавшись на рассылку Keras: [keras-users@googlegroups.com](mailto:keras-users@googlegroups.com).
- Вы также можете следить за мной в Twitter: [@fchollet](https://twitter.com/fchollet).

## ЗАКЛЮЧИТЕЛЬНОЕ СЛОВО

Вот и закончилось второе издание книги «Глубокое обучение на 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 неизвестного 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.

*Франсуа Шолле*

## **Глубокое обучение на Python. 2-е межд. издание**

*Перевел с английского А. Н. Киселев*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>К. Тарасевич</i>
Художник	<i>В. Мостицан</i>
Корректоры	<i>М. Молчанова, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».  
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,  
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2022.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,  
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,  
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 02.09.22. Формат 70×100/16. Бумага офсетная.  
Усл. п. л. 46,440. Тираж 1500. Заказ 0000.



**Джереми Ховард,  
Сильвейн Гуггер**

## **Глубокое обучение с fastai и PyTorch: минимум формул, минимум кода, максимум эффективности**

Обычно на глубокое обучение смотрят с ужасом, считая, что только доктор математических наук или ботан, работающий в крутой айтишной корпорации, могут разобраться в этой теме. Отбросьте стереотипы: любой программист, знакомый с Python, может добиться впечатляющих результатов. Как? С помощью fastai — библиотеки, предоставляющей комфортный интерфейс для решения наиболее популярных задач.

Создатели fastai доказали, что самые модные и актуальные приложения можно делать быстро и не засыпать над скучными теоретическими выкладками и зубодробительными формулами.



**Сергей Николенко,  
Артур Кадурин,  
Екатерина Архангельская**

## **Глубокое обучение**

Перед вами — первая книга о глубоком обучении, написанная на русском языке. Глубокие модели оказались ключом, который подходит ко всем замкам сразу: новые архитектуры и алгоритмы обучения, а также увеличившиеся вычислительные мощности и появившиеся огромные наборы данных, привели к революционным прорывам в компьютерном зрении, распознавании речи, обработке естественного языка и многих других типично «человеческих» задачах машинного обучения. Эти захватывающие идеи, вся история и основные компоненты революции глубокого обучения, а также самые современные достижения этой области, доступно и интересно изложены в книге. Максимум объяснений, минимум кода, серьезный материал о машинном обучении и увлекательное изложение — в этой уникальной работе замечательных российских ученых и интеллектуалов.