

# Глубокое обучение с R и Keras

Второе издание



Франсуа Шоме,  
создатель Keras

 MANNING

 ДМК  
ИЗДАТЕЛЬСТВО

# Глубокое обучение с R и Keras

Умение работать с моделями глубокого обучения стало важным навыком современных ученых, исследователей и программистов. API языка R для Keras и TensorFlow делает глубокое обучение доступным для всех пользователей R, даже если у них нет опыта работы с машинным обучением или нейронными сетями.

Интуитивно понятные объяснения, четкие иллюстрации и наглядные примеры помогут вам освоить основные навыки глубокого обучения с помощью R, такие как компьютерное зрение, обработка естественного языка, работа с текстом, и даже изучить передовую архитектуру Transformer. Это исправленное и расширенное издание является адаптацией второго издания книги Франсуа Шолле Deep Learning with Python («Глубокое обучение с Python»).

*Для читателей со средними навыками программирования на R. Опыт работы с Keras, TensorFlow или моделями глубокого обучения не требуется.*

## Основные темы книги:

- классификация и сегментация изображений;
- прогнозирование временных рядов;
- классификация текстов и машинный перевод;
- генерация текста, перенос стиля и генерация изображений.

Франсуа Шолле — инженер-программист в Google и создатель фреймворка Keras.

Томаш Калиновски — инженер-программист в RStudio и специалист по сопровождению пакетов R для Keras и Tensorflow.

Дж. Дж. Аллер — основатель RStudio и автор первого издания этой книги.

«Эта книга незаменима для ученых и инженеров, которые хотят расширить свои познания».

*Фернандо Гарсия Седано,  
Grupo Epelsa*

«Вам не найти лучшего учебника, если вы новичок в глубоком обучении или хотите развить свои навыки работы с R».

*Майкл Петри, Voxplot Analytics*

«Понятные иллюстрации и наглядные примеры будут полезны всем, от начинающих до опытных практиков глубокого обучения».

*Эдвард Ли,  
Йельский университет*

Интернет-магазин: [www.dmkpress.com](http://www.dmkpress.com)

Оптовая продажа: КТК «Галактика»  
[books@aliants-kniga.ru](mailto:books@aliants-kniga.ru)



ISBN 978-5-93700-189-4



Франсуа Шолле

# Глубокое обучение с R и Keras

# *Deep Learning with R*

SECOND EDITION

**FRANÇOIS CHOLLET**  
with **TOMASZ KALINOWSKI**  
and **J. J. ALLAIRE**



MANNING  
Shelter Island

# *Глубокое обучение с R и Keras*

**ФРАНСУА ШОЛЛЕ**  
при участии  
**Томаша Калиновски**  
и **Дж. Дж. Аллера**



Москва, 2023

УДК 004.85  
ББК 32.971.3  
Ш78

**Шолле Ф.**

Ш78 Глубокое обучение с R и Keras / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2022. – 646 с.: ил.

**ISBN 978-5-93700-189-4**

Прочитав эту книгу, вы получите четкое представление о том, что такое глубокое обучение, когда его следует применять и каковы его ограничения. Авторы описывают стандартный рабочий процесс поиска решения задачи машинного обучения и рассказывают, как устранять часто возникающие проблемы. Всесторонне рассматривается использование Keras для решения самых разнообразных прикладных задач, в числе которых классификация и сегментация изображений, прогнозирование временных рядов, классификация текста, машинный перевод, генерация текста и многое другое.

Издание адресовано читателям со средними навыками программирования на R. Опыт работы с Keras, TensorFlow или моделями глубокого обучения не требуется.

УДК 004.85  
ББК 32.971.3

Copyright © DMK Press 2022. Authorized translation of the English edition © 2022 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

## Отзывы о первом издании

«Самое понятное объяснение глубокого обучения, которое я когда-либо встречал... приятно и легко читается».

– *Ричард Тобиас, Cephasonics*

«Эта книга сокращает разрыв между идеями и работающей системой глубокого обучения».

– *Петр Рабинович, Akamai*

«Все основные темы и концепции глубокого обучения раскрыты и доходчиво объяснены с использованием примеров кода и графиков вместо математических формул».

– *Срджан Сантич, Springboard.com*

# Оглавление

---

1	■	Что такое глубокое обучение? .....	24
2	■	Математические основы нейронных сетей .....	55
3	■	Введение в Keras и TensorFlow.....	104
4	■	Примеры работы с нейросетью: классификация и регрессия .....	143
5	■	Основы машинного обучения .....	173
6	■	Обобщенный рабочий процесс машинного обучения .....	215
7	■	Работа с Keras: углубленные навыки.....	239
8	■	Глубокое обучение в компьютерном зрении .....	277
9	■	Глубокое обучение для компьютерного зрения.....	321
10	■	Глубокое обучение и временные ряды .....	370
11	■	Глубокое обучение в обработке текстов .....	408
12	■	Генеративные модели глубокого обучения .....	484
13	■	Глубокое обучение в реальной жизни .....	547
14	■	Заключение.....	571

# Содержание

---

Оглавление .....	6
Предисловие .....	15
Благодарности .....	17
Об этой книге .....	18
Об иллюстрации на обложке .....	22
Об авторах .....	23

<b>1</b>	<b>Что такое глубокое обучение?</b> .....	24
1.1	Искусственный интеллект, машинное и глубокое обучение .....	25
1.1.1	Искусственный интеллект .....	25
1.1.2	Машинное обучение .....	26
1.1.3	Извлечение правил и представлений из данных .....	28
1.1.4	«Глубина» глубокого обучения .....	31
1.1.5	Принцип действия глубокого обучения в трех рисунках .....	33
1.1.6	Каких успехов достигло глубокое обучение .....	35
1.1.7	Не верьте рекламной шумихе .....	36
1.1.8	Перспективы развития ИИ .....	37
1.2	Краткая история машинного обучения .....	38
1.2.1	Вероятностное моделирование .....	39
1.2.2	Первые нейронные сети .....	39
1.2.3	Ядерные методы .....	40
1.2.4	Деревья решений, случайные леса и градиентный бустинг .....	42
1.2.5	Назад к нейронным сетям .....	43
1.2.6	Отличительные черты глубокого обучения .....	44
1.2.7	Современный ландшафт машинного обучения .....	45
1.3	Почему глубокое обучение? Почему сейчас? .....	47
1.3.1	Оборудование .....	48
1.3.2	Данные .....	49
1.3.3	Алгоритмы .....	50
1.3.4	Новая волна инвестиций .....	51
1.3.5	Демократизация глубокого обучения .....	52
1.3.6	Ждать ли продолжения этой тенденции? .....	52

<b>2</b>	<b>Математические основы нейронных сетей</b> .....	55
2.1	Первое знакомство с нейронной сетью .....	56
2.2	Представление данных для нейронных сетей .....	60
2.2.1	Скаляры (тензоры нулевого ранга) .....	61
2.2.2	Векторы (тензоры первого ранга) .....	61
2.2.3	Матрицы (тензоры второго ранга) .....	62
2.2.4	Тензоры третьего и более высокого рангов .....	62
2.2.5	Ключевые атрибуты .....	62
2.2.6	Манипулирование тензорами в R .....	64
2.2.7	Пакеты данных .....	64
2.2.8	Практические примеры тензоров с данными .....	65
2.2.9	Векторные данные .....	65
2.2.10	Временные ряды, или последовательности данных .....	66
2.2.11	Изображения .....	67
2.2.12	Видеоданные .....	67
2.3	Шестеренки нейронных сетей: операции с тензорами .....	68
2.3.1	Поэлементные операции .....	69
2.3.2	Операции с тензорами разной размерности .....	70
2.3.3	Скалярное произведение тензоров .....	72
2.3.4	Изменение формы тензора .....	74
2.3.5	Геометрическая интерпретация операций с тензорами .....	75
2.3.6	Геометрическая интерпретация глубокого обучения .....	79
2.4	Механизм нейронных сетей: оптимизация на основе градиента .....	80
2.4.1	Что такое производная? .....	82
2.4.2	Производная операций с тензорами: градиент .....	83
2.4.3	Стохастический градиентный спуск .....	85
2.4.4	Объединение производных: алгоритм обратного распространения ошибки .....	88
2.5	Возвращаясь к нашему первому примеру .....	95
2.5.1	Повторная реализация нашего первого примера с нуля в TensorFlow .....	97
2.5.2	Выполнение одного шага обучения .....	99
2.5.3	Полный цикл обучения .....	101
2.5.4	Оценка модели .....	102
	Краткие итоги главы .....	103
<b>3</b>	<b>Введение в Keras и TensorFlow</b> .....	104
3.1	Что такое TensorFlow? .....	105
3.2	Что такое Keras? .....	106
3.3	Keras и TensorFlow: краткая история .....	107
3.4	Интерфейсы Python и R: краткая история .....	108
3.5	Настройка среды разработки для глубокого обучения .....	109
3.5.1	Установка Keras и TensorFlow .....	110
3.6	Первые шаги с TensorFlow .....	111
3.6.1	Тензоры TensorFlow .....	112
3.7	Атрибуты тензоров .....	113

3.7.1	Форма тензора и ее изменение .....	114
3.7.2	Срезы тензоров .....	116
3.7.3	Операции с тензорами разной размерности .....	117
3.7.4	Модуль <i>tf</i> .....	117
3.7.5	Неизменность тензоров и переменные .....	119
3.7.6	Математические операции в <i>TensorFlow</i> .....	120
3.7.7	Взгляд на <i>API GradientTape</i> с другой стороны .....	121
3.7.8	Полный пример: линейный классификатор в чистом <i>TensorFlow</i> .....	122
3.8	Анатомия нейронной сети и основы <i>API Keras</i> .....	127
3.8.1	Слои: строительные блоки глубокого обучения .....	128
3.8.2	От слоев к моделям .....	132
3.8.3	Этап «компиляции»: настройка процесса обучения .....	134
3.8.4	Выбор функции потерь .....	137
3.8.5	Использование метода <i>fit()</i> .....	138
3.8.6	Отслеживание потерь и показателей на контрольных данных .....	139
3.8.7	Использование модели после обучения .....	140
	Краткие итоги главы .....	141

## 4 Примеры работы с нейросетью: классификация и регрессия .....

4.1	Классификация отзывов к фильмам: пример бинарной классификации .....	145
4.1.1	Набор данных <i>IMDB</i> .....	145
4.1.2	Подготовка данных .....	147
4.1.3	Создание модели .....	148
4.1.4	Проверка вашего выбора .....	151
4.1.5	Использование обученной сети для прогнозирования на новых данных .....	154
4.1.6	Продолжаем эксперименты .....	155
4.1.7	Промежуточные итоги .....	155
4.2	Классификация новостных лент: пример многоклассовой классификации .....	156
4.2.1	Набор данных <i>Reuters</i> .....	156
4.2.2	Подготовка данных .....	158
4.2.3	Построение модели .....	158
4.2.4	Проверка модели .....	159
4.2.5	Предсказания на новых данных .....	161
4.2.6	Другой способ обработки меток и потерь .....	162
4.2.7	Важность использования достаточно больших промежуточных слоев .....	162
4.2.8	Дальнейшие эксперименты .....	163
4.2.9	Промежуточные итоги .....	163
4.3	Предсказание цен на дома: пример регрессии .....	164
4.3.1	Набор данных с ценами на жилье в Бостоне .....	164
4.3.2	Подготовка данных .....	165
4.3.3	Построение модели .....	165

4.3.4	Оценка качества модели методом $K$ -кратной перекрестной проверки .....	166
4.3.5	Выдача прогнозов на новых данных .....	171
4.3.6	Промежуточные выводы .....	171
	Краткие итоги главы.....	171

<b>5</b>	<b>Основы машинного обучения .....</b>	<b>173</b>
5.1	Обобщение – цель машинного обучения .....	173
5.1.1	Недообучение и переобучение .....	174
5.1.2	Базовые принципы обобщения в глубоком обучении .....	180
5.2	Оценка моделей машинного обучения .....	187
5.2.1	Наборы данных для обучения, проверки и контроля.....	187
5.2.2	Использование критериев, основанных на здравом смысле .....	191
5.2.3	Что следует помнить об оценке модели .....	192
5.3	Улучшение качества обучения модели .....	193
5.3.1	Настройка ключевых параметров градиентного спуска.....	193
5.3.2	Использование лучшей априорно обоснованной архитектуры .....	196
5.3.3	Увеличение емкости модели .....	197
5.4	Как улучшить обобщение .....	199
5.4.1	Подготовка набора данных .....	199
5.4.2	Конструирование признаков .....	200
5.4.3	Использование ранней остановки .....	202
5.4.4	Регуляризация модели .....	202
	Краткие итоги главы.....	213

<b>6</b>	<b>Обобщенный рабочий процесс машинного обучения .....</b>	<b>215</b>
6.1	Постановка задачи.....	217
6.1.1	Уточнение задачи .....	217
6.1.2	Получение исходных данных.....	219
6.1.3	Добейтесь понимания данных.....	223
6.1.4	Выберите меру успеха.....	224
6.2	Разработка модели.....	225
6.2.1	Подготовка данных .....	225
6.2.2	Выбор протокола оценки .....	227
6.2.3	Как превзойти простой базовый уровень.....	228
6.2.4	Масштабирование: разработка модели, способной к переобучению .....	229
6.2.5	Регуляризация и настройка модели.....	230
6.3	Развертывание модели .....	231
6.3.1	Представление модели заказчику.....	231
6.3.2	Передача модели заказчику .....	232
6.3.3	Мониторинг модели в рабочей среде.....	236
6.3.4	Поддержка и обновление модели.....	236
	Краткие итоги главы.....	237

<b>7</b>	<b>Работа с Keras: углубленные навыки</b> .....	239
7.1	Широкий спектр рабочих процессов Keras .....	240
7.2	Различные способы построения моделей Keras .....	240
7.2.1	<i>Sequential API</i> .....	241
7.2.2	<i>Functional API</i> .....	244
7.2.3	Создание подкласса класса <i>Model</i> .....	251
7.2.4	Смешивание и сочетание разных компонентов .....	255
7.2.5	Используйте правильные инструменты .....	256
7.3	Использование встроенных циклов обучения и оценки .....	256
7.3.1	Разработка собственных метрик.....	257
7.3.2	Использование обратных вызовов .....	260
7.3.3	Разработка собственных обратных вызовов .....	262
7.3.4	Мониторинг и визуализация с помощью <i>TensorBoard</i> .....	264
7.4	Разработка собственных циклов обучения и оценки .....	266
7.4.1	Обучение или логический вывод.....	267
7.4.2	Использование метрик на низком уровне.....	268
7.4.3	Полный цикл обучения и оценки .....	269
7.4.4	Увеличьте быстрдействие с помощью <i>tf.function()</i> .....	272
7.4.5	Использование <i>fit()</i> с пользовательским циклом обучения.....	273
	Краткие итоги главы.....	276
<b>8</b>	<b>Глубокое обучение в компьютерном зрении</b> .....	277
8.1	Введение в сверточные нейронные сети .....	278
8.1.1	Операция свертки .....	281
8.1.2	Выбор максимального значения из соседних ( <i>max-pooling</i> ).....	286
8.2	Обучение сверточной нейронной сети с нуля на небольшом наборе данных .....	289
8.2.1	Целесообразность глубокого обучения для решения задач с небольшими наборами данных .....	290
8.2.2	Загрузка данных.....	290
8.2.3	Построение сети.....	293
8.2.4	Предварительная обработка данных .....	295
8.2.5	Расширение данных .....	301
8.3	Использование предварительно обученной сверточной нейронной сети .....	305
8.3.1	Выделение признаков .....	306
8.3.2	Дообучение ранее обученной модели .....	316
	Краткие итоги главы.....	320
<b>9</b>	<b>Глубокое обучение для компьютерного зрения</b> .....	321
9.1	Три основные задачи компьютерного зрения.....	322
9.2	Пример сегментации изображения.....	323
9.3	Современные стандартные архитектуры сверточных сетей .....	333
9.3.1	Модульность, иерархия и повторное использование.....	334
9.3.2	Остаточные связи .....	337

9.3.3	Пакетная нормализация .....	341
9.3.4	Разделяемые по глубине свертки .....	344
9.3.5	Применим знания на практике: мини-модель, подобная Xception .....	347
9.4	Интерпретация знаний сверточной нейросети .....	350
9.4.1	Визуализация промежуточных активаций .....	351
9.4.2	Визуализация сетевых фильтров .....	357
9.4.3	Визуализация тепловых карт активации класса .....	363
	Краткие итоги главы .....	369

## 10 Глубокое обучение и временные ряды.....370

10.1	Различные виды задач временных рядов.....	370
10.2	Пример прогнозирования температуры.....	372
10.2.1	Подготовка данных.....	376
10.2.2	Простое решение задачи без привлечения машинного обучения.....	380
10.2.3	Решение с использованием базовой модели машинного обучения.....	382
10.2.4	Эксперимент с одномерной сверточной сетью .....	384
10.2.5	Первый вариант простой рекуррентной модели .....	387
10.3	Рекуррентные нейронные сети .....	388
10.3.1	Рекуррентный слой в Keras.....	391
10.4	Продвинутое применение рекуррентных нейронных сетей ...	396
10.4.1	Использование рекуррентного прореживания для борьбы с переобучением.....	397
10.4.2	Наложение рекуррентных слоев .....	400
10.4.3	Использование двунаправленных рекуррентных сетей .....	402
10.4.4	Что дальше .....	405
	Краткие итоги главы.....	407

## 11 Глубокое обучение в обработке текстов.....408

11.1	Обработка естественного языка: обзор отрасли .....	408
11.2	Подготовка текстовых данных.....	411
11.2.1	Стандартизация текста .....	412
11.2.2	Разделение текста (токенизация).....	413
11.2.3	Индексация словаря.....	414
11.2.4	Использование слоя <code>layer_text_vectorization</code> .....	416
11.3	Два подхода к представлению групп слов: наборы и последовательности .....	420
11.3.1	Подготовка данных обзоров фильмов IMDB.....	421
11.3.2	Обработка слов без учета порядка .....	424
11.3.3	Обработка последовательности слов .....	432
11.4	Архитектура Transformer.....	446
11.4.1	Механизм самовнимания .....	446
11.4.2	Многоголовое внимание .....	452
11.4.3	Кодировщик в архитектуре Transformer .....	453
11.4.4	Когда следует использовать модели последовательности, а не модели мешка слов.....	463

11.5	Помимо классификации текста: обучение преобразованию последовательностей.....	464
11.5.1	Пример машинного перевода .....	466
11.5.3	Рекуррентная модель преобразования последовательностей .....	469
11.5.4	Преобразование последовательностей с Transformer .....	476
	Краткие итоги главы.....	482
<b>12</b>	<b>Генеративные модели глубокого обучения .....</b>	<b>484</b>
12.1	Генерирование текста с помощью Keras.....	486
12.1.1	Краткая история генеративных сетей.....	486
12.1.2	Как генерируют последовательности данных?.....	488
12.1.3	Важность стратегии выбора .....	488
12.1.4	Реализация генерации текста с помощью Keras.....	491
12.1.5	Обратный вызов генерации текста с выборкой при разной температуре .....	495
12.1.6	Подведение итогов .....	502
12.2	DeepDream .....	502
12.2.1	Реализация DeepDream в Keras.....	503
12.2.2	Подведение итогов .....	511
12.3	Нейронный перенос стиля.....	512
12.3.1	Функция потерь содержания.....	513
12.3.2	Функция потерь стиля .....	513
12.3.3	Реализация переноса стиля в Keras.....	514
12.3.4	Подведение итогов .....	522
12.4	Генерация изображений с помощью вариационных автокодировщиков .....	522
12.4.1	Выбор шаблонов из скрытых пространств изображений .....	523
12.4.2	Концептуальные векторы для редактирования изображений .....	524
12.4.3	Вариационные автокодировщики.....	525
12.4.4	Реализация VAE с помощью Keras .....	528
12.4.5	Подведение итогов .....	534
12.5	Введение в генеративно-состязательные сети .....	534
12.5.1	Реализация генеративно-состязательной сети .....	536
12.5.2	Полезные технические приемы .....	537
12.5.3	Получение набора данных CelebA.....	538
12.5.4	Дискриминатор.....	540
12.5.5	Генератор.....	541
12.5.6	Состязательная сеть .....	542
12.5.7	Подведение итогов .....	545
	Краткие итоги главы.....	546
<b>13</b>	<b>Глубокое обучение в реальной жизни .....</b>	<b>547</b>
13.1	Получение максимальной отдачи от ваших моделей .....	548
13.1.1	Оптимизация гиперпараметров .....	548
13.1.2	Ансамблирование моделей.....	557
13.2	Масштабируемое обучение моделей.....	559

13.2.1	Ускорение обучения на GPU со смешанной точностью .....	560
13.2.2	Обучение модели на нескольких GPU .....	563
13.2.3	Обучение модели на TPU .....	568
	Краткие итоги главы .....	570

<b>14</b>	<b>Заключение .....</b>	<b>571</b>
14.1	Краткий обзор ключевых понятий .....	572
14.1.1	Различные подходы к ИИ .....	572
14.1.2	Что выделяет глубокое обучение среди других подходов к машинному обучению .....	573
14.1.3	Как правильно воспринимать глубокое обучение .....	573
14.1.4	Ключевые технологии глубокого обучения .....	575
14.1.5	Обобщенный рабочий процесс машинного обучения .....	576
14.1.6	Основные архитектуры сетей .....	577
14.1.7	Пространство возможностей .....	582
14.2	Ограничения глубокого обучения .....	584
14.2.1	Риск очеловечивания моделей глубокого обучения .....	585
14.2.2	Принципиальное различие между автоматом и интеллектом .....	587
14.2.3	Различие между локальным и экстремальным обобщением .....	589
14.2.4	Предназначение интеллекта .....	592
14.2.5	Восхождение по уровням обобщения .....	593
14.3	Курс на большую универсальность в ИИ .....	594
14.3.1	О важности постановки правильной цели: правило короткого пути .....	594
14.3.2	Новая цель .....	597
14.4	Реализация интеллекта: недостающие ингредиенты .....	599
14.4.1	Построение и использование абстрактных аналогий .....	599
14.4.2	Два полюса абстракции .....	601
14.4.3	Сочетание двух полюсов абстракции .....	604
14.4.4	Недостающая половина картинки .....	604
14.5	Будущее глубокого обучения .....	606
14.5.1	Модели как программы .....	606
14.5.2	Машинное обучение и синтез программ .....	608
14.5.3	Сочетание глубокого обучения и синтеза программ .....	608
14.5.4	Непрерывное обучение и повторное использование модульных подпрограмм .....	611
14.5.5	Долгосрочная перспектива .....	612
14.6	Как не отстать от прогресса в быстро развивающейся отрасли .....	614
14.6.1	Решения реальных задач на сайте Kaggle .....	614
14.6.2	Знакомство с последними разработками на сайте arXiv .....	614
14.6.3	Исследование экосистемы Keras .....	615
14.7	Заключительное слово .....	616
	Приложение. Введение в Python для пользователей R .....	617
	Предметный указатель .....	641

# Предисловие

---

Если вы решили приобрести эту книгу, то наверняка слышали о небывалом успехе методики глубокого обучения в области искусственного интеллекта. Мы прошли путь от почти бесполезного распознавания образов и речи до невероятно эффективного решения этих задач. Последствия такого внезапного прогресса отразились почти повсеместно. Сегодня мы применяем глубокое обучение для решения целого ряда важных задач в таких разных областях, как визуализация медицинских данных, сельское хозяйство, автономное вождение, образование, предотвращение стихийных бедствий и промышленное производство.

Тем не менее я считаю, что глубокое обучение все еще находится в зачаточном состоянии. Пока оно реализовало лишь малую часть своего потенциала. Со временем глубокое обучение проникнет в каждую область, где может принести пользу, – трансформация, которая займет не одно десятилетие.

Однако для того чтобы начать внедрение технологии глубокого обучения во все задачи, которые можно решить с ее применением, мы должны сделать ее доступной как можно большему числу людей, включая неспециалистов – то есть тех, кто не является инженером-исследователем или аспирантом. Чтобы раскрыть весь потенциал глубокого обучения, мы должны полностью демократизировать его. И сегодня я считаю, что мы находимся на пороге исторического перелома, когда глубокое обучение выходит из академических лабораторий и отделов исследований и разработок крупных технологических компаний, чтобы стать обыденной частью набора инструментов каждого разработчика, – очень похоже на историю веб-технологий в конце 1990-х. Сейчас почти любой желающий может создать веб-сайт или веб-приложение для своего бизнеса или сообщества, хотя

в 1998 году для этого потребовалась бы команда специалистов. В недалеком будущем любой, у кого есть идея и базовые навыки программирования, сможет создавать интеллектуальные приложения, которые обучаются на основе данных.

Когда в марте 2015 года я выпустил первую версию фреймворка глубокого обучения Keras, я не стремился сделать общедоступным искусственный интеллект (ИИ). Я несколько лет занимался исследованиями в области машинного обучения и создал Keras для использования в собственных экспериментах. Но с 2015 года в область глубокого обучения пришли сотни тысяч новичков; многие из них выбрали Keras в качестве любимого инструмента. Наблюдая за тем, как множество новичков и опытных специалистов используют Keras самыми неожиданными и эффективными способами, я пришел к выводу, что нужно задуматься о доступности и демократизации ИИ. Я осознал, что чем шире мы будем распространять эти технологии, тем ценнее они будут становиться. Доступность быстро стала одной из главных целей Keras, и за несколько лет сообществу разработчиков удалось добиться фантастических достижений в этом направлении. Мы «вручили» технологию глубокого обучения сотням тысяч людей, и они, в свою очередь, воспользовались ею для решения важных проблем, которые до недавнего времени считались неразрешимыми.

Книга, которую вы держите, – еще один шаг на пути к тому, чтобы сделать глубокое обучение доступным как можно большему количеству людей. Фреймворк Keras всегда нуждался в сопроводительном курсе, который одновременно освещал бы основы глубокого обучения, показывал примеры его использования и демонстрировал лучшие практики в применении Keras. В 2016 и 2017 годах я приложил изрядные усилия, что создать такой курс. Он лег в основу первого издания этой книги, выпущенной в декабре 2017 года. Книга быстро стала бестселлером по машинному обучению, разошлась тиражом более 50 000 экземпляров и была переведена на 12 языков.

Однако область глубокого обучения быстро развивается. С момента публикации первого издания произошло много важных событий – выпуск TensorFlow 2, растущая популярность архитектуры Transformer и многое другое. Поэтому в конце 2019 года я решил обновить свою книгу. Сначала я наивно думал, что смогу обойтись обновлением около 50 % контента, и объем второго издания почти не изменится. На самом деле после двух лет работы новая редакция оказалась более чем на треть длиннее, в ней 75 % нового материала. Это больше, чем обновление, это совершенно новая книга.

Я писал ее, стараясь максимально доступно объяснить идеи, лежащие в основе глубокого обучения и его реализации. Это не значит, что я преднамеренно упрощал изложение – я искренне убежден, что в теме глубокого обучения нет ничего сложного. Надеюсь, эта книга принесет вам пользу и поможет начать создавать интеллектуальные приложения и решать важные для вас задачи.

# Благодарности

---

Прежде всего я хочу поблагодарить сообщество Keras за помощь в создании этой книги. За последние шесть лет сообщество Keras выросло до сотен добровольных разработчиков исходного кода и более миллиона пользователей. Ваш вклад и отзывы помогли превратить Keras в то, чем он является сейчас.

Я хотел бы внести в эту благодарность глубоко личные нотки и поблагодарить свою жену за ее всемерную поддержку во время разработки Keras и написания этой книги.

Я также хочу поблагодарить Google за поддержку проекта Keras. Было очень приятно, когда Keras решили использовать в качестве высокоуровневого API TensorFlow. Бесшовная интеграция Keras и TensorFlow приносит большую пользу пользователям обоих продуктов и делает глубокое обучение доступным для широкого круга пользователей.

Я хочу поблагодарить сотрудников издательства Manning, благодаря которым эта книга стала возможной: издателя Марджана Бейса и всех сотрудников редакторского и производственного отделов, в том числе Майкла Стивенса, Дженнифер Стаут, Александра Драгосавлевича, Энди Маринковича, Памелу Хант, Сьюзен Ханиуэлл, Кери Хейлз, Пола Уэллса и многих других, работавших за кулисами.

Большое спасибо всем рецензентам – Арнальдо Аялу Мейеру, Давиде Кремонези, Дхинакарану Венкату, Эдварду Ли, Фернандо Гарсии Седано, Джоэлу Котарски, Марсио Николау, Майклу Петри, Питеру Хенстоку, Шахнавазу Али, Сураву Бисвасу, Тиаго Бритто Борхесу, Тони Дубицки, Владу Навицки и всем, кто присылал нам отзывы о первом издании книги. Ваши предложения помогли сделать эту книгу лучше.

Что касается технической стороны, особая благодарность Нино-славу Черкезу, который работал техническим корректором книги.

# Об этой книге

---

Эта книга предназначена для всех, кто хочет освоить глубокое обучение с нуля или расширить свои знания о глубоком обучении. Независимо от того, являетесь ли вы инженером по разработке систем машинного обучения, специалистом по данным или студентом университета, вы найдете для себя много полезного на страницах этой книги.

Вы будете изучать глубокое обучение наиболее эффективным способом – начиная с простых понятий, а затем переходя к самым современным методам. Вы убедитесь, что эта книга обеспечивает баланс между интуитивным знанием, теорией и практикой. Мы старались как можно меньше использовать математические формулы, предпочитая вместо этого объяснять основные идеи машинного и глубокого обучения с помощью подробных фрагментов кода и интуитивно понятных образных моделей. Из многочисленных примеров кода, снабженных подробными комментариями, практических рекомендаций и простых объяснений вы получите знания, которых достаточно, чтобы использовать глубокое обучение для решения прикладных задач.

В примерах кода мы используем платформу глубокого обучения Keras с TensorFlow 2 в качестве вычислительного движка. Примеры демонстрируют лучшие известные нам приемы использования Keras и TensorFlow 2 по состоянию на 2022 год.

Прочитав эту книгу, вы получите четкое представление о том, что такое глубокое обучение, когда его следует применять и каковы его ограничения. Вы познакомитесь со стандартным рабочим процессом поиска решения задачи машинного обучения, а также узнаете, как устранять часто возникающие проблемы. Вы научитесь использовать Keras для решения самых разнообразных прикладных задач, начиная с компьютерного зрения и заканчивая обработкой естест-

венного языка, – среди них классификация изображений, сегментация изображений, прогнозирование временных рядов, классификация текста, машинный перевод, генерация текста и многое другое.

## *Кому адресована эта книга*

Эта книга написана для людей с опытом программирования на R, желающих начать знакомство с темой машинного обучения с технологии глубокого обучения. Но она также может быть полезной и другим категориям читателей:

- если вы специалист по обработке и анализу данных, знакомый с машинным обучением, эта книга позволит вам получить достаточно полное практическое представление о глубоком обучении, наиболее быстро развивающемся направлении в области машинного обучения;
- если вы исследователь или прикладной специалист в области глубокого обучения, желающий освоить фреймворк Keras, вы найдете в этой книге лучший углубленный курс по Keras;
- если вы аспирант, изучающий технологии глубокого обучения в ходе обязательного курса, в этой книге вы найдете практическое дополнение к своим учебникам, которое поможет вам лучше понять принцип действия нейросетей и познакомит с наиболее эффективными приемами.

Даже люди с техническим складом ума, которые не занимаются программированием регулярно, найдут эту книгу полезной для знакомства с базовыми и продвинутыми понятиями глубокого обучения.

Для понимания примеров кода вам понадобится знание языка R на среднем уровне. Не обязательно иметь опыт работы с машинным или глубоким обучением: эта книга охватывает все необходимые основы с нуля. Не требуется также иметь какой-то особенной математической подготовки — вполне достаточно знания математики на уровне средней школы.

## *О примерах кода*

Эта книга содержит большое количество примеров исходного кода как в пронумерованных листингах, так и в виде обычного текста. В обоих случаях исходный код представлен шрифтом фиксированной ширины, чтобы он отличался от обычного текста. Вывод работающего кода аналогичным образом отформатирован шрифтом фиксированной ширины, но также снабжен вертикальной серой полосой слева. На протяжении всей книги вы найдете код и выходные данные кода, чередующиеся следующим образом:

```
print("R is awesome!")
```

| [1] "R is awesome!"

Во многих случаях исходный код был переформатирован; нам пришлось добавить разрывы строк и изменить отступы, чтобы код умещался на доступном пространстве страницы. В редких случаях этого было недостаточно, и некоторые листинги содержат маркеры продолжения строки (↪). Кроме того, многие комментарии к исходному коду были удалены из листингов, если код подробно описан в тексте. Многие листинги содержат дополнительные примечания, указывающие на важные нюансы кода.

Вы можете загрузить все примеры кода с сайта книги по адресу <https://livebook.manning.com/book/deep-learning-with-r-second-edition/> или из репозитория GitHub <https://github.com/t-kalinowski/deep-learning-with-R-2nd-edition-code>, а также с сайта издательства «ДМК Пресс» по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

## Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте [www.dmkpress.com](http://www.dmkpress.com), зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com); при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу [http://dmkpress.com/authors/publish\\_book/](http://dmkpress.com/authors/publish_book/) или напишите в издательство по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

## Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com). Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

## Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо

из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

## *Об иллюстрации на обложке*

---

Рисунок на обложке второго издания этой книги – «Образ китайки в 1700 году» – взят из книги Томаса Джеффериса, опубликованной между 1757 и 1772 годами. В то время по одежде людей было легко определить, где люди живут, чем зарабатывают на жизнь и каково их общественное положение. Издательство Manning стремится отразить изобретательность и инициативность компьютерного бизнеса при помощи книжных обложек, основанных на богатом разнообразии региональной культуры многовековой давности, оживленной портретами из таких коллекций, как эта.

# Об авторах

---

**Франсуа Шолле** (François Chollet) – создатель Keras, одного из наиболее популярных фреймворков глубокого обучения. В настоящее время он работает инженером-программистом в Google, где возглавляет команду Keras. Кроме того, он занимается исследованиями в области абстракции, рассуждений и способов достижения большей универсальности в искусственном интеллекте.

**Томаш Калиновски** (Tomasz Kalinowski) работает инженером-программистом в RStudio, где занимается сопровождением пакетов TensorFlow и Keras R. На предыдущих должностях он работал ученым и инженером, применяя машинное обучение к широкому спектру наборов данных и предметных областей.

**Дж. Дж. Аллер** (J. J. Allaire) – основатель RStudio и создатель интегрированной среды разработки RStudio IDE. Является автором интерфейса R к библиотекам TensorFlow и Keras.

# 1

## Что такое глубокое обучение?

---

### *Эта глава охватывает следующие темы:*

- обобщенные определения основных понятий;
- история развития машинного обучения;
- ключевые факторы роста популярности глубокого обучения и потенциал развития.

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

ние? Насколько это важно? В каком направлении пойдет дальнейшее развитие? Можно ли верить поднятой шумихе?

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

## 1.1 Искусственный интеллект, машинное и глубокое обучение

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



Рис. 1.1 Искусственный интеллект, машинное и глубокое обучение

### 1.1.1 Искусственный интеллект

Идея искусственного интеллекта появилась в 1950-х, когда группа энтузиастов из только зарождающейся области информатики задались вопросом, можно ли заставить компьютеры «думать», – вопросом, последствия которого мы изучаем до сих пор.

Хотя многие лежащие в основе ИИ идеи зародились в предшествующие годы и даже десятилетия, «искусственный интеллект» окончательно оформился как область исследований в 1956 году, когда Джон Маккарти, в то время молодой доцент кафедры математики в Дартмутском колледже, организовал летний семинар с весьма амбициозными задачами:

*Исследование должно опираться на гипотезу о том, что каждый аспект обучения или любое другое свойство интеллекта в принципе поддаются настолько точному описанию, что их можно будет смоделировать с помощью машины. Будут предприняты попытки найти способ, как заставить машины использовать язык, формировать абстракции и концепции, решать задачи, традиционно остающиеся уде-*

*лом людей, и улучшать себя. Мы думаем, что можно добиться значительного прогресса в решении одной или нескольких из этих проблем, если тщательно подобранная группа ученых будет работать над ней вместе в течение лета.*

В конце лета семинар завершился, а исчерпывающее решение задачи так и не удалось найти. Тем не менее на нем присутствовало много исследователей, которые впоследствии стали пионерами в области ИИ, и фактически он запустил интеллектуальную революцию, которая продолжается и по сей день.

Коротко эту область можно определить так: *автоматизация интеллектуальных задач, обычно выполняемых людьми*. Соответственно, ИИ – это область, охватывающая машинное обучение и глубокое обучение, а также включающая многие подходы, не связанные с обучением. Только представьте, что вплоть до 1980-х в большинстве книг про ИИ вообще не упоминалось 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 дать программисту достаточный набор явных правил для манипулирования знаниями. Этот подход известен как символический ИИ и являлся доминирующей парадигмой ИИ с 1950-х до конца 1980-х. Пик его популярности пришелся на бум *экспертных систем* в 1980-х.

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

### 1.1.2 Машинное обучение

Много лет назад в викторианской Англии жила леди Ада Лавлейс – друг и соратник Чарльза Бэббиджа, изобретателя аналитической вычислительной машины, первого известного механического компьютера. Несомненно, аналитическая машина опередила свое время, но она не задумывалась как универсальный компьютер, когда разрабатывалась в 1830-х и 1840-х, потому что идея универсальных вычислений еще не родилась. Эта машина просто давала возможность использовать механические операции для автоматизации некоторых вычислений из области математического анализа, что и обусловило такое ее название. Надо сказать, что аналитическая машина была интеллектуальным потомком более ранних попыток кодирования математических операций в форме шестерни, таких

как *паскалина* или ступенчатый счетный механизм Лейбница – усовершенствованная версия паскалины. Разработанная Блезом Паскалем в 1642 году (в возрасте 19 лет!), паскалина была первым в мире механическим калькулятором, который мог складывать, вычитать, умножать и даже делить числа.

В 1843 году Ада Лавлейс заметила:

*Аналитическая машина не может создавать что-то новое. Она может делать все, что мы и сами знаем, как выполнять... ее цель состоит лишь в том, чтобы помогать нам выполнять то, с чем мы уже хорошо знакомы.*

Даже с учетом 179-летней исторической перспективы наблюдения леди Лавлейс не перестает наводить на размышления. Может ли компьютер общего назначения «создать» что-нибудь или он всегда будет вынужден тупо выполнять процессы, которые мы, люди, полностью понимаем? Родятся ли у него когда-нибудь оригинальные мысли? Может ли он учиться на собственном опыте? Может ли он проявить креативность?

Позднее пионер ИИ Алан Тьюринг в своей знаменитой статье «Computing Machinery and Intelligence»<sup>1</sup> назвал это замечание «аргументом Ады Лавлейс». В этой статье был представлен тест Тьюринга, а также перечислены основные идеи, которые могут привести к созданию ИИ<sup>2</sup>. Тьюринг пришел к выводу – очень провокационному на то время, – что в принципе компьютеры способны имитировать все аспекты человеческого интеллекта.

Обычный способ заставить компьютер выполнять полезную работу – это попросить человека-программиста написать *правила* – компьютерную программу, которой нужно следовать, чтобы преобразовать входные данные в соответствующие ответы, точно так же, как леди Лавлейс записывала пошаговые инструкции для аналитической машины. В машинном обучении люди вводят данные и ответы, соответствующие этим данным, а на выходе получают правила. Эти правила затем можно применить к новым данным для получения оригинальных ответов. В машинном обучении система *обучается*, а не программируется явно. Ей передаются многочисленные примеры, имеющие отношение к решаемой задаче, а она находит в этих примерах статистическую структуру, которая позволяет системе выработать правила для автоматического решения задачи (рис. 1.2). Например, чтобы автоматизировать задачу определения фотографий, сделанных в отпуске, можно передать системе машинного обучения множество примеров фотографий, уже классифици-

<sup>1</sup> A. M. Turing. *Computing Machinery and Intelligence*. Mind 59, no. 236 (1950): 433–460.

<sup>2</sup> Хотя тест Тьюринга часто воспринимают как буквальный вызов, которую должен достичь ИИ, сам Тьюринг просто использовал его как концептуальный прием в философской дискуссии о природе познания.

рованных людьми, и система изучит статистические правила классификации конкретных фотографий.



Рис. 1.2 Машинное обучение: новая парадигма программирования

Расцвет машинного обучения начался только в 1990-х, но эта область быстро превратилась в наиболее популярный и успешный раздел ИИ, и эта тенденция была подкреплена появлением более быстродействующей аппаратуры и огромных наборов данных. Машинное обучение тесно связано с математической статистикой, но имеет несколько важных отличий. В отличие от статистики, машинное обучение обычно имеет дело с большими и сложными наборами данных (например, состоящими из миллионов фотографий, каждая из которых состоит из десятков тысяч пикселей), к которым практически невозможно применить классические методы статистического анализа, такие как байесовские методы. Как результат машинное и в особенности глубокое обучение имеют ограниченную математическую базу – возможно, слишком ограниченную – и основываются почти исключительно на инженерных решениях. В отличие от теоретической физики или математики, машинное обучение – это прикладная область, основанная на эмпирических данных и сильно зависящая от достижений в области программного и аппаратного обеспечения.

### 1.1.3 Извлечение правил и представлений из данных

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

- *входные данные* – например, если решается задача распознавания речи, такими входными данными могут быть файлы с записью речи разных людей. Если решается задача классификации изображений, такими данными могут быть изображения;
- *примеры ожидаемых результатов* – в задаче распознавания речи это могут быть транскрипции звуковых файлов, составленные

людьми. В задаче классификации изображений ожидаемым результатом могут быть теги, такие как «собака», «кошка» и др.;

- *способ оценки качества работы алгоритма* – это необходимо для определения, насколько далеко отклоняются результаты, возвращаемые алгоритмом, от ожидаемых. Оценка используется как сигнал обратной связи для корректировки работы алгоритма. Этот этап корректировки мы и называем обучением.

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

Прежде чем двинуться дальше, давайте определим, что есть представление данных. По сути, это другой способ взглянуть на данные – иначе их представить или закодировать. Например, цветное изображение можно закодировать в формате RGB (red-green-blue – красный–зеленый–синий) или HSV (hue-saturation-value – тон–насыщенность–значение): это два разных представления одних и тех же данных. Некоторые задачи трудно решаются с данными в одном представлении, но легко – в другом. Например, задача «выбрать все красные пиксели в изображении» легче решается с данными в формате RGB, тогда как задача «сделать изображение менее насыщенным» проще решается с данными в формате HSV. Главная задача моделей машинного обучения как раз и заключается в поиске соответствующего представления входных данных – преобразований, которые сделают данные более пригодными для решения задачи, такой как классификация.

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

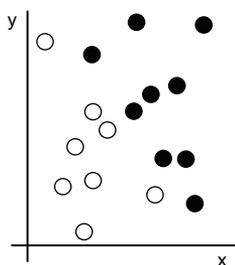


Рис. 1.3 Пример некоторых данных

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 мы располагаем следующими данными:

- входными данными являются координаты точек;
- ожидаемым результатом является цвет точек;
- мерой качества 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 других, могло бы быть изменение системы координат, как показано на рис. 1.4.



Рис. 1.4 Изменение системы координат

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

В этом случае мы определили изменение координат вручную – мы использовали наш человеческий интеллект, чтобы придумать собственное подходящее представление данных. Это нормально для такой чрезвычайно простой задачи, но сможете ли вы сделать то же самое, если перед вами стоит задача классификации изображений рукописных цифр? Сможете ли вы записать явные, исполняемые компьютером преобразования изображений, которые четко выделяют различия между 6 и 8 или между 1 и 7 для любого почерка?

В какой-то степени это возможно. Правила, основанные на представлениях цифр, таких как «количество замкнутых циклов» или вертикальные и горизонтальные пиксельные гистограммы, могут неплохо различать рукописные цифры. Но поиск таких полезных представлений вручную – тяжелая работа, и, как вы легко догадаетесь, полученная система, основанная на подобных правилах, очень неустойчива. Поддерживать работоспособность такой системы – сущий кошмар. Каждый раз, когда вы сталкиваетесь с новым примером почерка, нарушающим ваши тщательно продуманные правила, вам придется добавлять новые преобразования данных и новые правила, при этом учитывая их взаимодействие с каждым ранее добавленным правилом.

Наверняка вы уже подумали, что этот сложный и болезненный процесс нужно автоматизировать. Что, если мы попытаемся целенаправленно искать различные наборы автоматически сгенерированных представлений данных и основанных на них правил и выявлять лучшие из них, используя в качестве обратной связи процент правильно классифицированных цифр в некотором наборе эталонных данных? Так вот, это и есть машинное обучение! Под машинным обучением мы понимаем процесс автоматического поиска преобразований данных, которые создают полезные представления, руководствуясь некоторым сигналом обратной связи, – такие представления, которые поддаются более простым правилам, решающим поставленную задачу.

В качестве таких преобразований могут выступать изменения координат (как в нашем примере с классификацией точек) или получение гистограммы пикселей и циклов обхода (как в нашем примере с классификацией цифр), но также это могут быть линейные проекции, переносы, нелинейные операции (например, правило «выбрать все точки такие, что  $x > 0$ ») и т. д. Алгоритмы машинного обучения обычно не очень изобретательны в поиске этих преобразований; они просто просматривают predetermined набор операций, называемый *пространством гипотез*. Например, пространство всех возможных изменений координат будет нашим пространством гипотез в примере классификации точек.

Иными словами, машинное обучение – это поиск значимого представления некоторых входных данных в predetermined пространстве возможностей с использованием сигнала обратной связи. Эта простая идея позволяет решать чрезвычайно широкий круг интеллектуальных задач: от распознавания речи до автоматического вождения автомобиля. Теперь, когда вы получили представление о том, что понимается под *обучением*, давайте посмотрим, что особенного в *глубоком обучении*.

### 1.1.4 «Глубина» глубокого обучения

Глубокое обучение – это особый раздел машинного обучения: новый подход к поиску представления данных, делающий упор на изучение последовательных слоев (или уровней) все более значимых представлений. Под «глубиной» в глубоком обучении не подразумевают более глубокое понимание, достигаемое этим подходом; идея заключается в многослойном представлении. Количество слоев, на которые делится модель данных, называют *глубиной* модели. Другими подходящими названиями для этой области машинного обучения могли бы служить: *многослойное обучение* и *иерархическое обучение*. Современные модели глубокого обучения часто состоят из десятков и даже сотен последовательных слоев представления, и все они автоматически определяются под воздействием обучающих данных. Кроме того, существуют другие подходы к машинному обучению,

ориентированные на изучение одного-двух слоев представления данных; по этой причине их иногда называют *поверхностным обучением* (shallow learning).

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

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

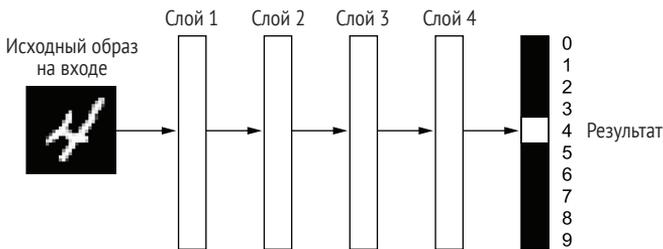


Рис. 1.5 Глубокая нейронная сеть для классификации цифр

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

С технической точки зрения глубокое обучение – это многоступенчатый способ получения представления данных. Идея проста,

но, как оказывается, очень простые механизмы в определенном масштабе могут выглядеть непонятными и таинственными.

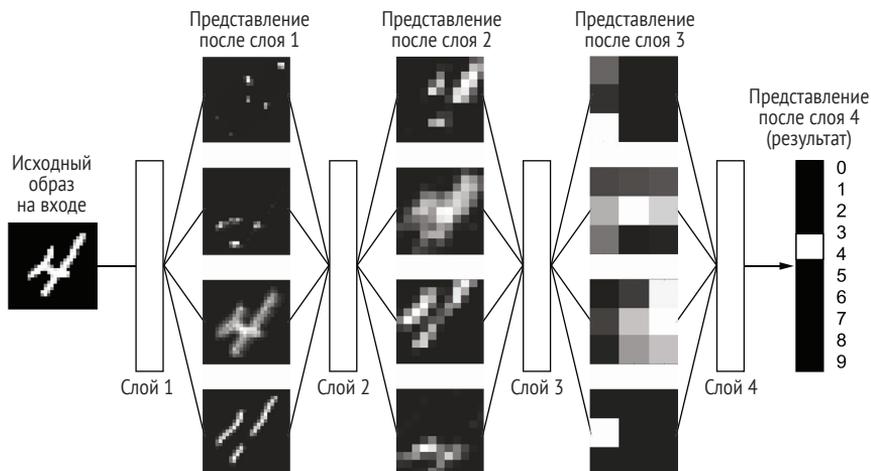


Рис. 1.6 Глубокие представления, получаемые моделью классификации цифр

### 1.1.5 Принцип действия глубокого обучения в трех рисунках

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

Что именно уровень делает со своими входными данными, определяется его весами, которые фактически являются набором чисел. Выражаясь техническим языком, можно сказать, что преобразование, реализуемое слоем, *параметризуется* его весами (рис. 1.7). (Весы также иногда называют *параметрами* слоя.) В данном контексте под *обучением* подразумевается поиск набора значений весов всех слоев в сети, при котором сеть будет правильно отображать образцовые входные данные в соответствующие им результаты. Но есть одна проблема: глубокая нейронная сеть может содержать 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.

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

торуую также называют *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).

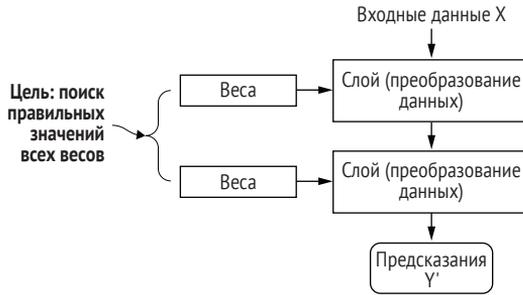


Рис. 1.7 Нейронная сеть параметризуется ее весами

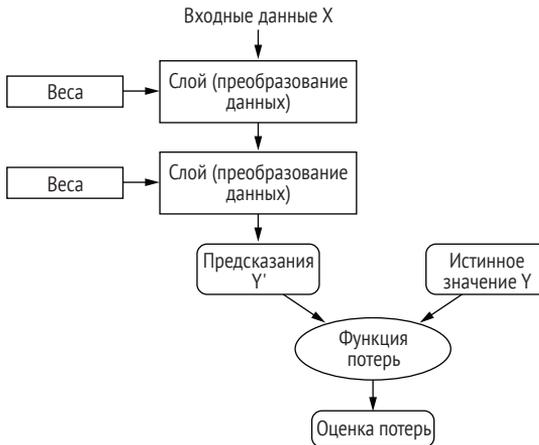


Рис. 1.8 Функция потерь оценивает качество результатов, производимых нейронной сетью

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

Первоначально весам сети присваиваются случайные значения, то есть фактически сеть реализует последовательность случайных преобразований. Естественно, получаемый ею результат далек от идеала, и оценка потерь, соответственно, очень 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 и возвращающая результаты, близкие к истинным, называется обученной сетью. Повторюсь еще раз: это простой механизм, который при определенном масштабе начинает выглядеть непонятным и таинственным.

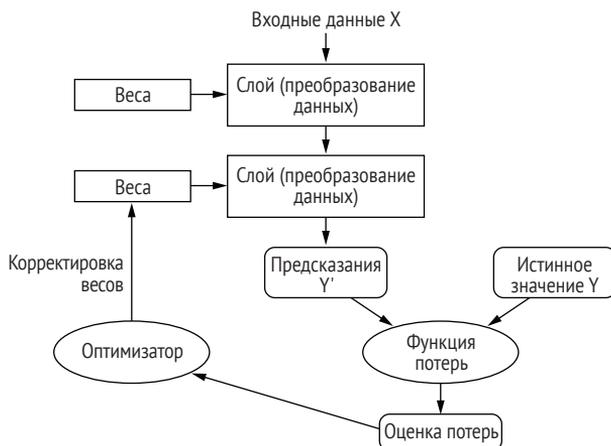


Рис. 1.9 Оценка потерь используется как обратная связь для корректировки весов

### 1.1.6 Каких успехов 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 настоящая революция, с особенно заметными успехами в моделировании восприятия и даже обработки естественного языка – задачах, кажущихся естественными и понятными для человека, но долгое время не дававшихся компьютерам. В частности, глубокое обучение достигло прорывов в следующих областях, традиционно сложных для машинного обучения:

- классификация изображений на уровне человека;
- распознавание речи на уровне человека;
- распознавание рукописного текста на уровне человека;
- улучшение качества машинного перевода с одного языка на другой;
- улучшение качества машинного чтения текста вслух;
- цифровые помощники, такие как Google Now и Amazon Alexa;
- управление автомобилем почти на уровне человека;

- повышение точности целевой рекламы, используемой компаниями Google, Baidu и Bing;
- повышение релевантности поиска в интернете;
- появление возможности отвечать на вопросы, заданные на разговорном языке;
- игра в го лучше человека.

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

Мы все еще не оценили возможности глубокого обучения в полной мере. Мы начали с большим успехом применять его для решения широкого круга задач, которые еще несколько лет назад казались неразрешимыми, – автоматически расшифровывать десятки тысяч древних манускриптов, хранящихся в Апостольском архиве Ватикана, обнаруживать и классифицировать болезни растений в полевых условиях с помощью простого смартфона, помогать онкологам и рентгенологам интерпретировать данные медицинских изображений, прогнозировать стихийные бедствия, такие как наводнения, ураганы или даже землетрясения, и т. д. С каждой новой вехой мы приближаемся к эпохе, когда глубокое обучение будет применяться во всех областях человеческой деятельности – в науке, медицине, производстве, энергетике, транспорте, разработке программного обеспечения, сельском хозяйстве и даже художественном творчестве.

### **1.1.7** *Не верьте рекламной шумихе*

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 применения, такие как автопилоты для автомобилей, находятся практически на заключительной стадии реализации, многие другие, такие как полноценные диалоговые системы, перевод между произвольными языками на уровне человека и понимание естественного языка на уровне человека, скорее всего, еще долгое время будут оставаться недостижимыми. В частности, не стоит всерьез воспринимать разговоры об интеллекте на уровне человека. Завышенные ожидания от ближайшего будущего таят опасность: из-за невозможности реализации новых технологий инвестиции в исследования будут падать и прогресс замедлится на долгое время.

Такое уже происходило раньше. В прошлом ИИ пережил две волны подъема оптимизма, за которыми следовал спад, сопровождаемый разочарованиями и скептицизмом и, как результат, снижением

финансирования. Все началось с символического ИИ в 1960-х. В те ранние годы давались весьма многообещающие прогнозы развития ИИ. Один из самых известных пионеров и сторонников подхода символического ИИ Марвин Мински 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 три–восемь лет у нас появится машина с интеллектом среднего человека». В 2023 году это достижение все еще кажется далеким будущим – пока мы не можем предсказать, сколько времени уйдет на это, – но в 1960-х и в начале 1970-х некоторые эксперты (как и многие люди ныне) полагали, что ИИ находится прямо за углом. Несколько лет спустя, из-за неоправдавшихся высоких ожиданий, исследователи и правительственные фонды отвернулись от этой области – так началась первая зима ИИ (вполне уместная метафора, потому что все это происходило вскоре после начала холодной войны).

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

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

### 1.1.8 *Перспективы развития ИИ*

Даже притом, что наши ожидания на ближайшую перспективу могут быть нереалистичными, долгосрочная картина выглядит весьма ярко. Мы только начинаем применять глубокое обучение к решению многих важных задач: от медицинских диагнозов до цифровых помощников. В последние пять лет исследования в области ИИ продвигались удивительно быстро, во многом благодаря высокому уровню финансирования, никогда прежде не наблюдавшемуся в короткой истории искусственного интеллекта, но пока слишком малому, чтобы этот прогресс воплотить в продукты и процессы, формирующие наш мир. Большинство результатов исследований в глубоком обучении пока не нашли практического применения, по крайней мере применения к решению полного спектра задач. Ваш доктор и ваш бухгалтер пока не используют ИИ. Вы сами в своей по-

вседневной жизни тоже, вероятно, не используете технологии ИИ. Конечно, вы можете задавать простые вопросы своему смартфону и получать разумные ответы, вы можете получить весьма полезные рекомендации при выборе товаров на Amazon.com и по фразе «день рождения» быстро найти в Google Photos фотографии со дня рождения вашей дочери, который был в прошлом месяце. Это, несомненно, большой шаг вперед. Но такие инструменты лишь дополняют нашу жизнь. ИИ еще не занял центральное место в нашей жизни, работе и мыслях.

Сейчас трудно поверить, что ИИ может оказать значительное влияние на наш мир, потому что еще не развернулся в полной мере, – так в 1995 году трудно было поверить в будущее влияние интернета. В то время большинство не понимало, какое отношение к ним может иметь интернет и как он изменит их жизнь. То же можно сегодня сказать о глубоком обучении и об искусственном интеллекте. Будьте уверены: эра искусственного интеллекта 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 к миру, который все более усложняется и наполняется информацией. И что особенно важно, ИИ будет способствовать человечеству в движении вперед, помогая ученым делать новые прорывные открытия во всех областях науки, от геномики до математики.

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

Не верьте рекламной шумихе, 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, но ИИ придет и изменит наш мир самым невероятным образом.

## **1.2** *Краткая история машинного обучения*

Глубокое обучение достигло уровня общественного внимания и инвестиций, невиданных прежде в истории искусственного интеллекта, но это не первая успешная форма машинного обучения. Можно с уверенностью сказать, что большинство алгоритмов машинного обучения, используемых сейчас в промышленности, не являются алгоритмами глубокого обучения. Глубокое обучение не всегда является правильным инструментом: иногда просто недостаточно данных

для глубокого обучения, иногда проблема лучше решается с применением других алгоритмов. Если глубокое обучение – ваш первый контакт с машинным обучением, вы можете оказаться в ситуации, когда, получив в руки молоток глубокого обучения, вы начнете воспринимать все задачи машинного обучения как гвозди. Единственный способ не попасть в эту ловушку – познакомиться с другими подходами и практиковать их, когда это необходимо.

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

### 1.2.1 *Вероятностное моделирование*

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

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

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

### 1.2.2 *Первые нейронные сети*

Ранние версии нейронных сетей были полностью вытеснены современными вариантами, о которых рассказывается на страницах этой

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

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

### 1.2.3 Ядерные методы

По мере привлечения внимания исследователей к нейронным сетям в 1990-х и благодаря первому успеху приобрел известность новый подход к машинному обучению, быстро отправивший нейронные сети обратно в небытие: ядерные методы (kernel methods). *Ядерные методы* – это группа алгоритмов классификации, из которых наибольшую известность получил *метод опорных векторов* (Support Vector Machine, SVM). Современная формулировка SVM была предложена Владимиром Вапником и Коринной Кортес в начале 1990-х в Bell Labs и опубликована в 1995 году<sup>1</sup>. Прежняя линейная формулировка была опубликована Вапником и Алексеем Червоненкисом (Alexey Chervonenkis) в начале 1963 года<sup>2</sup>.

Метод опорных векторов предназначен для решения задач классификации путем поиска хороших решающих границ (рис. 1.10), разделяющих два набора точек, принадлежащих разным категориям. Поиск таких границ метод опорных векторов осуществляет в два этапа.

- 1 Данные отображаются в новое пространство более высокой размерности, где граница может быть представлена как гиперплоскость (если данные были двумерными, как на рис. 1.10, гиперплоскость вырождается в линию).
- 2 Хорошая решающая граница (разделяющая гиперплоскость)

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

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

вычисляется путем максимизации расстояния от гиперплоскости до ближайших точек каждого класса, этот этап называют максимизацией зазора. Это позволяет обобщить классификацию новых образцов, не принадлежащих обучающему набору данных.

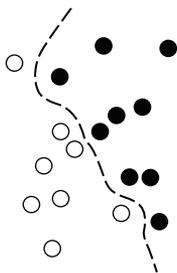


Рис. 1.10 Решающая граница

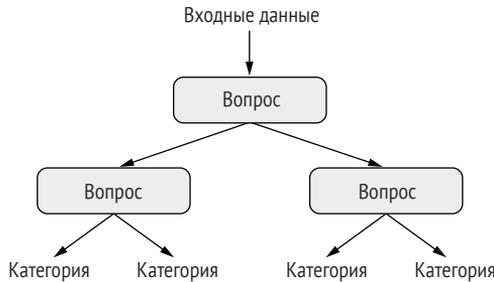
Методика отображения данных в пространство более высокой размерности, где задача классификации становится проще, может хорошо выглядеть на бумаге, но на практике часто оказывается трудноразрешимой. Вот тут и приходит на помощь изящная процедура *kernel trick* (ключевая идея, по которой ядерные методы получили свое название). Суть ее заключается в следующем: чтобы найти хорошие решающие гиперплоскости в новом пространстве, явно вычислять координаты точек в этом пространстве не требуется; достаточно вычислить расстояния между парами точек, что можно эффективно сделать с помощью функции ядра. Функция ядра – это незатратная вычислительная операция, 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 и поддающихся серьезному математическому анализу, что сделало его хорошо понятным и легко интерпретируемым. Благодаря этим свойствам метод опорных векторов приобрел чрезвычайную популярность на долгие годы.

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

## 1.2.4 Деревья решений, случайные леса и градиентный бустинг

*Деревья решений* – это иерархические структуры, которые позволяют классифицировать входные данные или предсказывать выходные значения по заданным исходным значениям (рис. 1.11). Они легко визуализируются и интерпретируются. Деревья решений, формируемые на основе данных, заинтересовали исследователей в 2000-х, и к 2010 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, вопрос «Коэффициент 2 в данных больше 3,5?»

В частности, алгоритм *случайного леса* (random forest) предложил надежный и практичный подход к обучению на основе деревьев решений, включающий создание большого количества специализированных деревьев решений с последующим объединением выдаваемых ими результатов. Случайные леса применимы к широкому кругу задач – можно сказать, что они почти всегда являются оптимальным алгоритмом для любых задач поверхностного машинного обучения. Когда в 2010 году был запущен популярный конкурсный веб-сайт Kaggle (<http://kaggle.com>), посвященный машинному обучению, случайные леса быстро превратились в наиболее популярную платформу, пока в 2014 году не появился метод градиентного бустинга. Метод градиентного бустинга, во многом напоминающий случайный лес, – это прием машинного обучения, основанный на объединении слабых моделей прогнозирования, обычно – деревьев решений. Он использует градиентный бустинг, способ улучшения любой модели машинного обучения путем итеративного обучения новых моделей, специализированных для устранения слабых мест в предыдущих моделях. Применительно к деревьям решений прием градиентного бустинга позволяет получить модели, которые в большинстве случаев превосходят случайные леса – при сохранении аналогичных свойств. На сегодняшний день это один из лучших алгоритмов, хотя и не самый лучший, для решения задач, не связанных с распознава-

нием. Наряду с глубоким обучением это один из наиболее широко используемых приемов в конкурсах на сайте Kaggle.

### 1.2.5 Назад к нейронным сетям

Примерно в 2010 году, несмотря на почти полную потерю интереса к нейронным сетям со стороны научного сообщества, ряд исследователей, продолжавших работать с нейронными сетями, стали добиваться важных успехов: группы Джеффри Хинтона из университета в Торонто, Йошуа Бенжю из университета в Монреале, Яна Лекуна из университета в Нью-Йорке 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 применением глубоких нейронных сетей, обучаемых на GPU, – это был первый практический успех современного глубокого обучения. Но перелом произошел в 2012 году, когда группа Хинтона приняла участие в ежегодном соревновании ImageNet по крупномасштабному распознаванию образов. ImageNet предложила очень сложное на то время задание, заключающееся в классификации цветных изображений с высоким разрешением на 1000 разных категорий после обучения по выборке, включающей 1,4 млн 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 (Alex Krizhevsky), в которой советником был Джеффри Хинтон (Geoffrey Hinton), достигла точности в 83,6 % – значительный прорыв. С тех пор каждый год первые позиции в этом соревновании занимают глубокие сверточные нейронные сети. В 2015 году победитель достиг точности в 96,4 %, и задача классификации на ImageNet была сочтена решенной полностью.

Начиная с 2012 года глубокие сверточные нейронные сети (convolutional neural networks, сокращенно *convnet*) перешли в разряд передовых алгоритмов для всех задач распознавания образов; в более общем плане они с успехом могут использоваться в любых задачах распознавания. На крупных конференциях по распознаванию образов, проводившихся в 2015 и 2016 годах, было трудно найти презентацию, не включающую сверточные нейросети в том или ином виде. В то же время глубокое обучение нашло применение во многих других видах задач, таких как обработка естественного языка. Оно полностью заменило метод опорных векторов и деревья решений в широком круге задач. Например, в течение нескольких лет Европейская организация по ядерным исследованиям (European Organization for Nuclear Research, CERN) использовала методы на основе

---

<sup>1</sup> Показатель «точность первой пятерки» измеряет, как часто модель выбирает правильный ответ как одну из своих пяти догадок (из 1000 возможных ответов, в случае ImageNet).

деревьев решений для данных, получаемых с детектора частиц ATLAS в большом адронном коллайдере, но затем в CERN было принято решение перейти на использование глубоких нейронных сетей на основе Keras из-за их более высокой производительности и простоты обучения на больших наборах данных.

### 1.2.6 Отличительные черты глубокого обучения

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

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

Вы можете спросить: если суть вопроса заключается в получении нескольких последовательных слоев представлений, можно ли многократно применить методы поверхностного обучения для имитации эффекта глубокого обучения? На практике наблюдается быстрое уменьшение последовательных применений методов поверхностного обучения, поскольку оптимальный слой первого представления в трехслойной модели не является оптимальным первым слоем в однослойной или двухслойной модели. Особенность преобразования в глубоком обучении состоит в том, что модель может обучать все слои представления *вместе* и одновременно, а не последовательно (этот тип обучения также называют «жадным»). При совместном изучении признаков, когда модель корректирует один

из своих внутренних признаков, все прочие признаки, зависящие от него, автоматически корректируются в соответствии с изменениями, без вмешательства человека. Все контролируется единственным сигналом обратной связи: каждое изменение в модели служит конечной цели. Это намного более мощный подход, чем жадно накладывать поверхностные модели друг на друга, потому что позволяет изучать более сложные абстрактные представления, разбивая их на длинные ряды промежуточных пространств (слоев), в которых каждое последующее пространство получается в результате простого преобразования предыдущего.

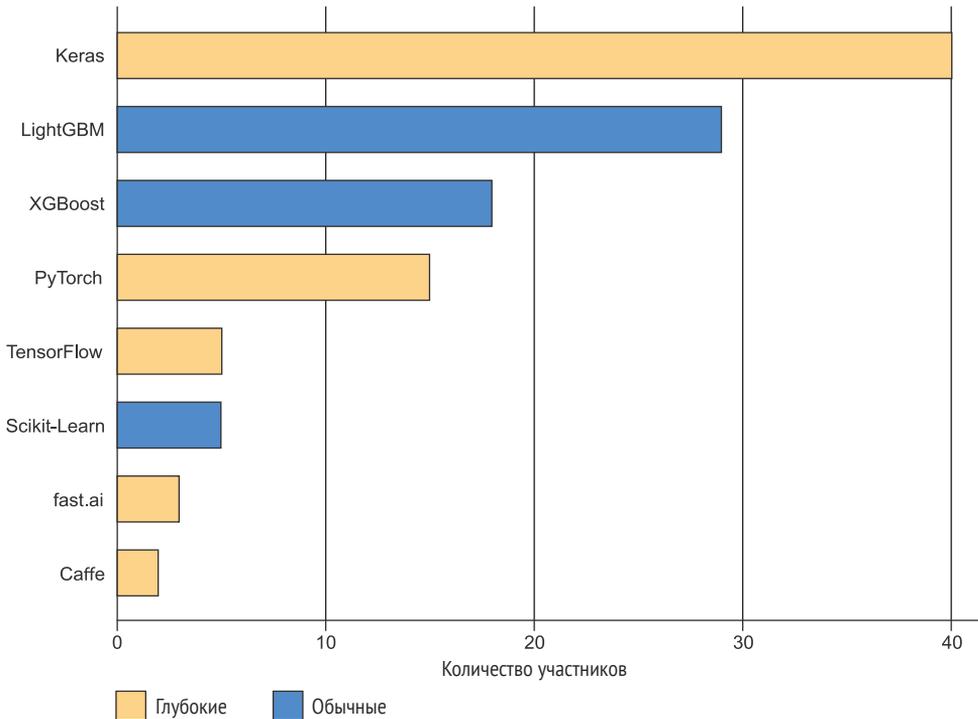
Методика глубокого обучения обладает двумя важными характеристиками: она *поэтапно, послойно конструирует все более сложные представления и совместно исследует промежуточные представления*, благодаря чему каждый слой обновляется в соответствии с потребностями представления слоя выше и потребностями слоя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

## 1.2.7 Современный ландшафт машинного обучения

Отличный способ получить представление о текущей ситуации с алгоритмами и инструментами машинного обучения – посмотреть соревнования по машинному обучению на Kaggle. Благодаря острой конкуренции (в некоторых конкурсах участвуют тысячи участников, а призовой фонд исчисляется миллионами долларов) и большому количеству охватываемых задач машинного обучения на сайте Kaggle можно реально оценить, какие подходы используются и насколько успешно. Так какой же алгоритм уверенно выигрывает состязания? Какими инструментами пользуются победители?

В начале 2019 года на сайте Kaggle провели опрос, в ходе которого команды, вошедшие в пятерку лучших в соревнованиях с 2017 года, спросили, какой основной программный инструмент они использовали для выполнения конкурсных заданий (рис. 1.12). Оказывается, лучшие команды склонны использовать либо методы глубокого обучения (чаще всего с помощью библиотеки Keras), либо градиентный бустинг над решающими деревьями (библиотеки LightGBM или XGBoost).

Речь не только о победителях соревнований. Kaggle также ежегодно проводит опрос среди специалистов в области машинного обучения и науки о данных по всему миру. Этот опрос, в котором приняли участие десятки тысяч респондентов, является одним из самых надежных источников информации о состоянии отрасли. На рис. 1.13 показан процент использования различных программных платформ машинного обучения.



**Рис. 1.12** Наиболее популярные инструменты машинного обучения, применяемые в конкурсах Kaggle

С 2016 по 2020 год во всей отрасли машинного обучения и науки о данных доминировали два подхода: глубокое обучение и градиентный бустинг над решающими деревьями. В частности, второй подход используется для задач, где доступны структурированные данные, тогда как глубокое обучение используется для задач восприятия, таких как классификация изображений.

Пользователи градиентного бустинга над решающими деревьями обычно применяют Scikit-Learn, XGBoost или LightGBM. В свою очередь, большинство практиков глубокого обучения используют Keras, часто в сочетании с родительским фреймворком TensorFlow. Общим для этих инструментов является то, что все они доступны в виде библиотек для языков R или Python, которые на сегодняшний день наиболее широко применяются в машинном обучении и обработке данных.

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

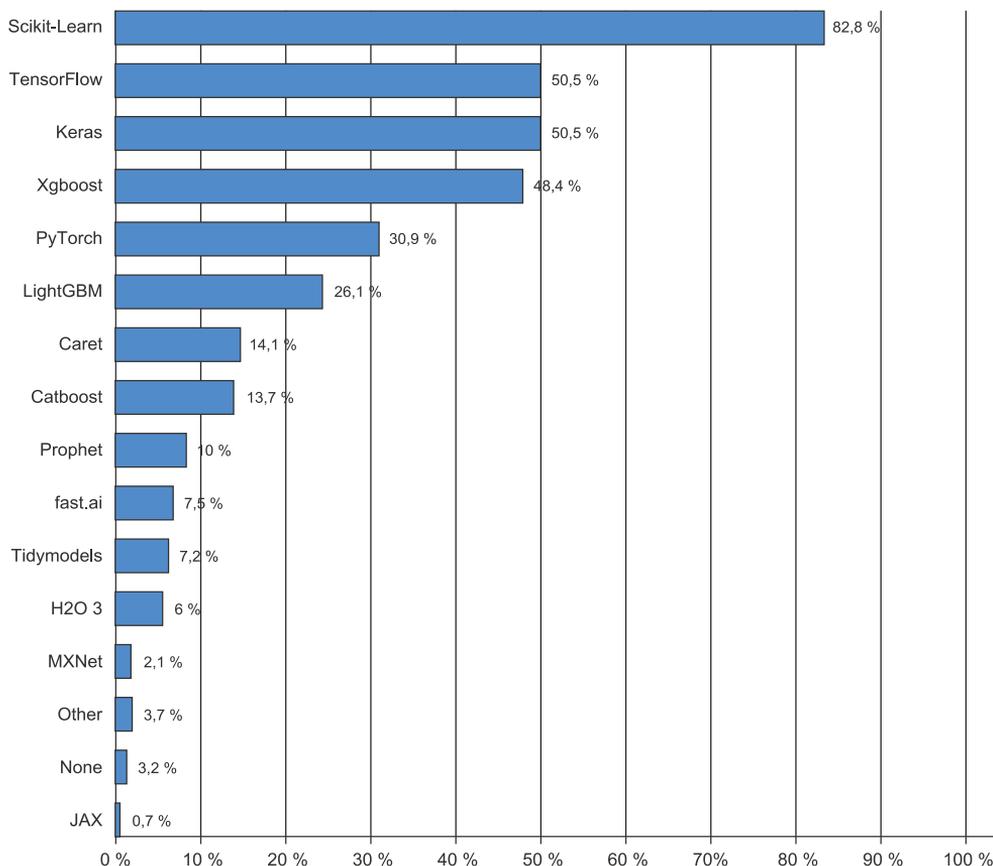


Рис. 1.13 Использование инструментов в машинном обучении и науке о данных (источник: <https://www.kaggle.com/kaggle-survey-2020>)

## 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 1990 году. Алгоритм *долгой краткосрочной памяти* (Long Short-Term Memory, LSTM), составляющий основу глубокого обучения для прогнозирования временных рядов, был предложен в 1997 году и с тех пор почти не изменился. Так почему же глубокое обучение начало применяться только с 2012 года? Что изменилось за эти два десятилетия? В целом машинное обучение стоит на «трех китах»:

- оборудование;
- наборы данных и тесты;
- современные алгоритмы.

Поскольку в области машинного обучения главную роль играют экспериментальные выводы, а не теория, алгоритмические достижения возможны только при наличии данных и оборудования, пригодных для проверки идей (или, как это часто бывает, для возрождения старых идей). Машинное обучение – это не математика и не физика, где прорывы могут быть сделаны с помощью ручки и бумаги. Это инженерная наука.

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

### 1.3.1 Оборудование

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

Но типичные модели глубокого обучения, используемые для распознавания образов или речи, требуют вычислительной мощности на порядки больше, чем мощность ноутбука. В течение 2000-х такие компании, как NVIDIA и AMD, вложили миллионы долларов в разработку быстрых процессоров с массовым параллелизмом (графических процессоров – Graphical Processing Unit, GPU) для поддержки графики все более реалистичных видеоигр – недорогих специализированных суперкомпьютеров, предназначенных для отображения на экране сложных трехмерных сцен в режиме реального времени. Эти инвестиции принесли пользу научному сообществу, когда в 2007 году компания NVIDIA выпустила CUDA (<https://developer.nvidia.com/about-cuda>), программный интерфейс для линейки своих GPU. Несколько GPU теперь могут заменить мощные кластеры на обычных процессорах в различных задачах, допускающих возможность массового распараллеливания вычислений, начиная с физического моделирования. Глубокие нейронные сети, выполняющие в основном умножение множества маленьких матриц, также допускают высокую степень распараллеливания; и ближе к 2011 году некоторые исследователи начали писать CUDA-реализации нейронных сетей. Одними из первых стали Дэн Кайесан<sup>1</sup> и Алекс Крижевски<sup>2</sup>.

Получилось так, что игровая индустрия субсидировала создание суперкомпьютеров для следующего поколения приложений искусственного интеллекта. Иногда крупные достижения начинают-

---

<sup>1</sup> *Flexible, High Performance Convolutional Neural Networks for Image Classification*, Proceedings of the 22nd International Joint Conference on Artificial Intelligence (2011), <http://mng.bz/nNOK>.

<sup>2</sup> *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in Neural Information Processing Systems 25 (2012), <http://mng.bz/2286>.

ся с игр. Современный графический процессор NVIDIA TITAN RTX, стоивший 2500 долларов США в конце 2019 года, способен выдать пиковую производительность 16 терафлопс с одинарной точностью (16 трлн операций в секунду с числами типа float32). Это почти в 500 раз больше производительности лучшего суперкомпьютера 1990-х Intel Touchstone Delta. Графическому процессору TITAN RTX требуется всего несколько часов для обучения модели ImageNet типа той, что выиграла конкурс ILSVRC в 2012 или 2013 году. Крупные компании обучают модели глубокого обучения на кластерах, состоящих из сотен GPU.

Более того, индустрия глубокого обучения начинает выходить за рамки GPU и инвестировать средства в развитие еще более специализированных процессоров, наиболее эффективно показывающих себя в области глубокого обучения. В 2016 году на своей ежегодной конференции Google I/O компания Google продемонстрировала свой проект тензорного процессора (Tensor Processing Unit, TPU): разработанная с нуля архитектура позволяет обучать глубокие нейронные сети значительно быстрее и гораздо эффективнее с точки зрения энергопотребления, чем лучшие графические процессоры. В 2020 году третья версия карты TPU имела вычислительную мощность 420 терафлопс. Это в 10 000 раз больше, чем у Intel Touchstone Delta в 1990 году.

Карты TPU предназначены для сборки в крупномасштабные модули, называемые «стручками» (pod). Один модуль (1024 карты TPU) достигает пиковой производительности 100 петафлопс. Для сравнения, это около 10 % пиковой вычислительной мощности самого крупного на сегодняшний день суперкомпьютера IBM Summit в национальной лаборатории Ок-Ридж, который состоит из 27 000 графических процессоров NVIDIA и достигает пиковой производительности около 1,1 экзафлопса.

### 1.3.2 Данные

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

То же можно сказать о видеороликах на YouTube. А Википедия стала ключевым источником наборов данных для задач обработки естественного языка.

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

Как показывает пример Kaggle, публичные конкурсы начиная с 2010 года – отличный способ мотивации исследователей и инженеров преодолевать все новые и новые рубежи. Наличие общих критериев оценки достижений исследователей значительно помогло недавнему росту глубокого обучения.

### 1.3.3 Алгоритмы

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

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

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

Только когда эти улучшения позволили создавать модели с 10 слоями и более, началось развитие глубокого обучения. Наконец в 2014, 2015 и 2016 годах были открыты еще более совершенные способы распространения градиента, такие как пакетная нормализация, остаточные связи и отдельные свертки по глубине.

Сегодня мы можем обучать с нуля модели произвольной глубины. Это открыло возможность использования чрезвычайно больших моделей, обладающих значительной *репрезентативностью*, то есть кодирующих очень богатые пространства гипотез. Такая невероятная масштабируемость является одной из определяющих характеристик

---

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

современного глубокого обучения. Архитектуры крупномасштабных моделей с десятками уровней и десятками миллионов параметров легли в основу важных достижений как в компьютерном зрении (например, такие архитектуры, как ResNet, Inception или Xception), так и в обработке естественного языка (например, архитектуры на основе трансформеров, такие как BERT, GPT-3 или XLNet).

### 1.3.4 Новая волна инвестиций

Как мы уже говорили, в 2012–2013 годах глубокое обучение вывело на новый 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 (рис. 1.14).

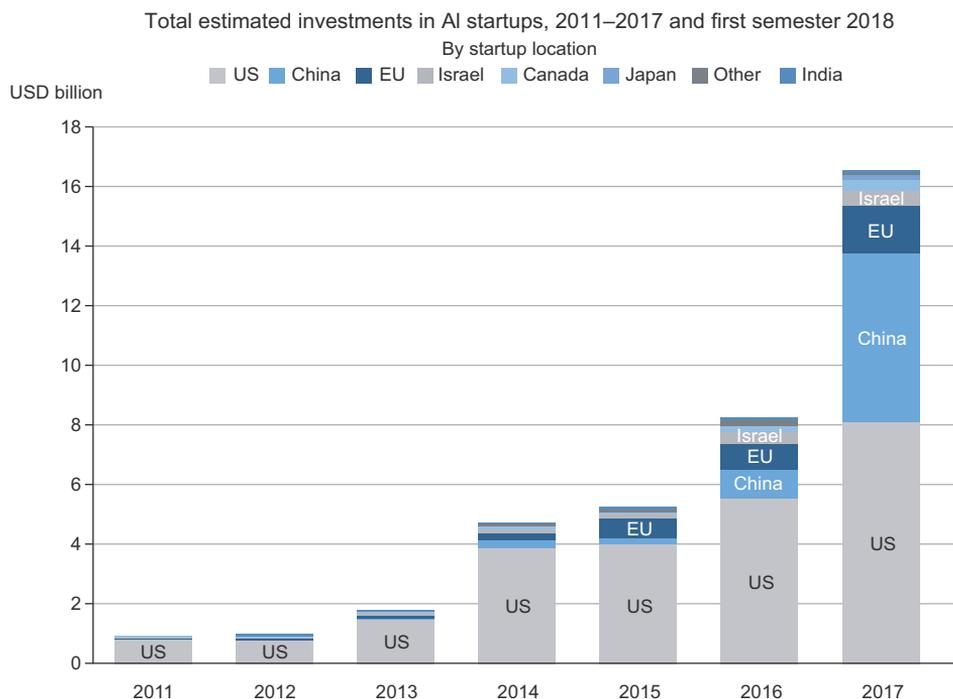


Рис. 1.14 Объем общих инвестиций в стартапы ИИ по данным ОЭСР (источник: <http://mng.bz/zGN6>)

В 2011 году, как раз перед тем, как в центре внимания оказалось глубокое обучение, общие венчурные инвестиции в ИИ по всему миру составили менее миллиарда долларов, которые почти полностью пошли на практическое применение поверхностных подходов машинного обучения. В 2015 году инвестиции выросли до более чем 5 млрд долларов, а в 2017 году – до ошеломляющих 16 млрд

долларов. За эти несколько лет появились сотни стартапов, пытающихся извлечь выгоду из поднявшейся шумихи. Между тем крупные компании, такие как Google, Facebook, Baidu и Microsoft, инвестировали деньги в исследования, проводившиеся внутренними подразделениями, и объемы этих инвестиций почти наверняка превысили инвестиции венчурного капитала.

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

В результате этой волны инвестиций за прошедшие 10 лет число людей, работающих над глубоким обучением, увеличилось с нескольких сотен до десятков тысяч, а прогресс в исследованиях достиг небывалого уровня.

### **1.3.5** *Демократизация глубокого обучения*

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

В настоящее время базовых навыков написания сценариев R или Python достаточно для проведения масштабных исследований в области глубокого обучения. В первую очередь это стало возможным благодаря появлению библиотеки TensorFlow – фреймворка для работы с тензорами, который поддерживает автодифференциацию, что значительно упрощает реализацию новых моделей, – а также удобных для пользователя библиотек, таких как Keras, которые делают глубокое обучение таким же простым, как использование кубиков LEGO. После выпуска в начале 2015 года Keras быстро стал популярным решением для глубокого обучения среди большого количества новых стартапов, аспирантов и исследователей, работающих 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, делающих инвестиции, и для исследователей? Или глубокое обучение – это просто увлечение, которое не продлится долго? Будем ли мы использовать глубокие нейронные сети через 20 лет?

---

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

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

- *простота* – глубокое обучение избавляет от необходимости конструировать признаки, заменяя сложные, противоречивые и тяжелые конвейеры простыми обучаемыми моделями, которые обычно строятся с использованием пяти-шести тензорных операций;
- *масштабируемость* – глубокое обучение легко поддается распараллеливанию на GPU или TPU, поэтому оно в полной мере может использовать закон Мура. Кроме того, обучение моделей можно производить итеративно, на небольших пакетах данных, что дает возможность проводить обучение на наборах данных произвольного размера. (Единственным узким местом является объем доступной вычислительной мощности для параллельных вычислений, которая, как следует из закона Мура, является быстро приближающимся барьером);
- *гибкость и пригодность к многократному использованию* – в отличие от многих предыдущих подходов, модели глубокого обучения могут обучаться на дополнительных данных без полного перезапуска, что делает их пригодными для непрерывного и продолжительного обучения – очень важное свойство для очень больших промышленных моделей. Кроме того, обучаемые модели глубокого обучения можно перенацеливать и, соответственно, использовать многократно: например, модель, обученную классификации изображений, можно включить в конвейер обработки видео. Это позволяет использовать предыдущие наработки для создания все более сложных и мощных моделей. Это также позволяет 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 год 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 первом издании этой книги в 2016 году я предсказывал, что глубокое обучение все еще находится в первой половине этой сигмоиды, а в следующие несколько лет нас ждет гораздо более революционный прогресс. Это подтвердилось на практике: в 2017 и 2018 годах

появились модели глубокого обучения на основе Transformer для обработки естественного языка, которые стали революцией в этой области, и в то же время глубокое обучение продолжало демонстрировать новые успехи в компьютерном зрении и распознавании речи. Сегодня, в 2022 году, глубокое обучение, похоже, вошло во вторую половину этой сигмоиды. Мы все еще можем ожидать значительного прогресса в ближайшие годы, но мы, вероятно, вышли из начальной фазы взрывного роста.

Сегодня меня очень вдохновляют примеры использования технологии глубокого обучения для решения бесчисленного перечня задач. Глубокое обучение все еще находится в стадии становления, и потребуется много лет, чтобы полностью раскрыть его потенциал.

# Математические основы нейронных сетей

---

## **Эта глава охватывает следующие темы:**

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

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

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

После этой главы у вас сформируется понимание основных теоретических принципов, лежащих в основе глубокого обучения, и вы будете готовы приступить к изучению Keras и TensorFlow в главе 3.

## 2.1 Первое знакомство с нейронной сетью

Рассмотрим конкретный пример нейронной сети, созданной с помощью пакета Keras R, которая обучается классификации рукописных цифр. Если у вас нет опыта использования Keras или других подобных библиотек, возможно, вы не все поймете в этом первом примере. Может быть, вы еще не установили Keras; в этом нет ничего страшного. В следующей главе мы рассмотрим каждый элемент в примере и подробно объясним их. Поэтому не волнуйтесь, если какие-то шаги покажутся вам непонятными или похожими на магию – мы ведь должны с чего-то начать.

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



Рис. 2.1 Образцы изображений MNIST

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

Не пытайтесь воспроизвести этот пример на своем компьютере прямо сейчас. Чтобы его опробовать, нужно сначала установить библиотеку Keras, о чем рассказывается в разделе 3.5.1. Набор данных MNIST входит состав в Keras в виде набора из четырех массивов R, организованных в два списка с именами `train` и `test`.

### Листинг 2.1 Загрузка набора данных MNIST в Keras

```
library(tensorflow)
library(keras)
```

```
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

Здесь `train_images` и `train_labels` – это обучающий набор, то есть данные, необходимые для обучения модели. После обучения модель будет проверяться тестовым (или контрольным) набором: `test_images` и `test_labels`. Изображения хранятся в массивах R, а метки – в одномерном массиве цифр от 0 до 9. Изображения и метки находятся в прямом соответствии один к одному. Давайте посмотрим, как выглядят обучающие данные:

```
> str(train_images)
| int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
```

```
> str(train_labels)
| int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
```

и контрольные данные:

```
> str(test_images)
| int [1:10000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
```

```
> str(test_labels)
| int [1:10000(1d)] 7 2 1 0 4 1 4 9 5 9 ...
```

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

Теперь построим сеть, как показано в следующем листинге. Помните, что от вас никто не ждет понимания всего и сразу.

### Листинг 2.2 Архитектура сети

```
model <- keras_model_sequential(list(
  layer_dense(units = 512, activation = "relu"),
  layer_dense(units = 10, activation = "softmax")
))
```

Основным строительным блоком нейронных сетей является *слой* (layer) – модуль обработки данных, который можно рассматривать как фильтр для данных. Он принимает некоторые данные и выводит их в более полезной форме. В частности, слои извлекают из подаваемых в них данных *представления*, которые, как мы надеемся, будут полезны для решаемой задачи. Фактически методика глубокого

обучения заключается в объединении простых слоев, реализующих определенную форму поэтапной очистки данных. Модель глубокого обучения можно сравнить с ситом, состоящим из последовательности фильтров все более тонкой очистки данных – слоев.

В данном случае наша сеть состоит из последовательности двух слоев Dense, которые являются тесно связанными (их еще называют *полносвязными*) нейронными слоями. Второй (и последний) уровень – это *слой классификации* с функцией softmax (softmax layer) с десятью параметрами, возвращающий массив с 10 оценками вероятностей (в сумме дающих 1). Каждая оценка определяет вероятность принадлежности текущего изображения к одному из 10 классов цифр.

Чтобы подготовить сеть к обучению, нужно настроить еще три параметра для этапа компиляции, показанного в листинге 2.3:

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

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

Обратите внимание, что мы не сохраняем возвращаемое значение функции `compile()`, поскольку модель модифицируется по месту.

Перед обучением мы выполним предварительную обработку данных, преобразовав их в форму, которую ожидает получить нейронная сеть, и масштабируем их так, чтобы все значения оказались в интервале  $[0, 1]$ . Предварительно исходные данные были сохранены в трехмерном массиве  $(60000, 28, 28)$  типа `integer`, значениями в котором являются числа в интервале  $[0, 255]$ . Мы преобразуем его в массив  $(60000, 28 * 28)$  типа `double` со значениями в интервале между 0 и 1.

### Листинг 2.4 Подготовка данных изображения

```
train_images <- array_reshape(train_images, c(60000, 28 * 28))
train_images <- train_images / 255
```

```
test_images <- array_reshape(test_images, c(10000, 28 * 28))
test_images <- test_images / 255
```

Обратите внимание, что для изменения формы массива мы использовали функцию `array_reshape()` вместо `dim()`. Причину мы объясним ниже, когда будем говорить об изменении формы тензора.

Теперь можно начинать обучение сети, для чего в случае использования библиотеки Keras достаточно вызвать метод `fit()` – он пытается подогнать (`fit`) модель под обучающие данные.

### Листинг 2.5 Обучение модели

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

```
Epoch 1/5
60000/60000 [=====] - 5s - 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 модели на обучающих данных и точность модели на обучающих данных. В данном случае мы быстро достигли точности 0,989 (98,9 %) на обучающих данных.

Теперь, когда у нас есть обученная модель, мы можем попробовать спрогнозировать с ее помощью вероятности классов для *новых* цифр – изображений, которые не были частью обучающих данных, таких как изображения из тестового набора.

### Листинг 2.6 Использование модели для прогнозирования

```
test_digits <- test_images[1:10, ]
predictions <- predict(model, test_digits)
str(predictions)
```

```
num [1:10, 1:10] 3.10e-09 3.53e-11 2.55e-07 1.00 8.54e-07 ...
```

```
predictions[1, ]
```

```
[1] 3.103298e-09 1.175280e-10 1.060593e-06 4.761311e-05 4.189971e-12
[6] 4.062199e-08 5.244305e-16 9.999473e-01 2.753219e-07 3.826783e-06
```

Каждое число индекса `i` в массиве `predictions[1, ]` соответствует вероятности того, что изображение цифры `test_digits[1, ]` принадлежит классу `i`. Эта первая проверочная цифра имеет самый высокий показатель вероятности (0,9999473, почти 1) для индекса 8, поэтому, согласно нашей модели, предсказана цифра 7 (потому что мы начинаем считать с 0):

```
which.max(predictions[1, ])
```

```
[1] 8
```

```
predictions[1, 8]
```

```
| [1] 0.9999473
```

Посмотрим, чему соответствует метка:

```
test_labels[1]
```

```
| [1] 7
```

Интересно, насколько хорошо наша модель классифицирует изображения цифр в среднем? Давайте проверим, вычислив среднюю точность по всему контрольному набору.

### Листинг 2.7 Проверка модели на новых данных

```
metrics <- evaluate(model, test_images, test_labels)
metrics["accuracy"]
```

```
| accuracy
0.9795
```

Точность на контрольном наборе составила 97,8 % – немного меньше, чем на обучающем наборе. Эта разница между точностью на обучающем и контрольном наборах демонстрирует пример *переобучения* (overfitting), когда модели машинного обучения показывают худшую точность на новом наборе данных по сравнению с обучающим.

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 R. Далее в этой и следующей главе я подробнее расскажу обо всех деталях, которые мы видели в этом примере, и поясню происходящее за кулисами. Вы узнаете о тензорах, объектах хранения данных в сети; операциях с тензорами, образующих слои сети; градиентном спуске, позволяющем вашей сети обучаться на учебных примерах.

## 2.2 Представление данных для нейронных сетей

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

По своей сути тензор – это контейнер для данных (обычно числовых), поэтому мы рассматриваем его как контейнер для чисел. Возможно, вы уже знакомы с матрицами, которые являются тензорами второго ранга; тензоры – это обобщение матриц с произвольным количеством *измерений* (обратите внимание, что в терминологии тензоров измерения часто называют *осями*).

В языке R тензоры реализованы в виде объектов `aggau`, созданных с помощью `base::aggau()`. В этом разделе мы займемся определением понятий, связанных с тензорами, поэтому будем использовать массивы R. Позже в главе 3 мы представим еще одну реализацию тензоров (`Tensor` в библиотеке `Tensorflow`).

### 2.2.1 Скаляры (тензоры нулевого ранга)

Тензор, содержащий единственное число, называется *скаляром* (скалярным, или нульмерным, тензором). В R нет типа данных для представления скаляров (все числовые объекты являются векторами, матрицами или массивами), но вектор, всегда имеющий длину 1, концептуально похож на скаляр.

### 2.2.2 Векторы (тензоры первого ранга)

Одномерный массив чисел называют *вектором*, или одномерным тензором (первого ранга). Одномерный тензор имеет единственную ось. Так выглядит пример вектора:

```
x <- as.array(c(12, 3, 6, 14, 7))
str(x)
```

```
| num [1:5(1d)] 12 3 6 14 7
```

```
length(dim(x))
```

```
| [1] 5
```

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 может иметь любое количество значений на каждой из них). Мерность может обозначать или количество элементов на данной оси (как в случае с пятимерным вектором), или количество осей в тензоре (как в пятимерном тензоре), что иногда может вызывать путаницу. В последнем случае технически более корректно говорить о *тензоре с рангом 5* (ранг тензора совпадает с количеством осей), но, в любом случае, для тензоров принято использовать неоднозначное обозначение: *пятимерный тензор*.

### 2.2.3 Матрицы (тензоры второго ранга)

Массив векторов – это матрица, или двумерный тензор (второго ранга). Матрица имеет две оси (часто их называют *строками* и *столбцами*). Матрицу можно рассматривать как прямоугольную таблицу с числами:

```
x <- array(seq(3 * 5), dim = c(3, 5))
x

|      [,1] [,2] [,3] [,4] [,5]
| [1,]   1   4   7  10  13
| [2,]   2   5   8  11  14
| [3,]   3   6   9  12  15
|
| dim(x)
| [1] 3 5
```

### 2.2.4 Тензоры третьего и более высокого рангов

Если упаковать такие матрицы в `dim`, получится трехмерный тензор, который можно визуально представить как числовой куб или столбик двумерных тензоров:

```
x <- array(seq(2 * 3 * 4), dim = c(2, 3, 4))
str(x)

| int [1:2, 1:3, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
|
| length(dim(x))
| [1] 3
```

Упаковав трехмерные тензоры, вы получите четырехмерный тензор, и т. д. В глубоком обучении чаще всего используются тензоры рангом от 0 до 4, но иногда, например при обработке видеоданных, дело может дойти и до пятимерных тензоров.

### 2.2.5 Ключевые атрибуты

Тензор определяется следующими тремя ключевыми атрибутами:

- *количество осей (ранг)* – например, трехмерный тензор имеет три оси, а матрица – две;
- *форма* – вектор целых чисел, описывающих количество измерений на каждой оси тензора. Например, матрица в предыдущем примере имеет форму (3, 5), а трехмерный тензор имеет форму (2, 3, 4). Вектор имеет форму с единственным элементом, например (5). Массивы R не различают одномерные векторы

и скалярные тензоры, но концептуально тензоры также могут быть скалярными с формой ( );

- *тип данных* – соответствует типу данных, содержащихся в тензоре. Например, тензор может иметь тип `integer` или `double`. Массивы R поддерживают встроенные типы данных R, такие как `double` и `integer`. Концептуально, однако, тензоры могут хранить любой однородный тип данных, и другие реализации тензоров также обеспечивают поддержку таких типов, как `float16`, `float32`, `float64` (соответствует `double` в R), `int32` (`integer` в R) и т. д. В TensorFlow вы также можете встретить тензоры типа `string`.

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

```
library(keras)
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

Узнаем количество осей тензора `train_images`:

```
length(dim(train_images))
```

```
[1] 3
```

его форму:

```
dim(train_images)
```

```
[1] 60000 28 28
```

и тип данных:

```
typeof(train_images)
```

```
[1] "integer"
```

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

Попробуем отобразить пятую цифру из этого трехмерного тензора (рис. 2.2).

### Листинг 2.8 Отображение пятой цифры

```
digit <- train_images[5, , ]
plot(as.raster(abs(255 - digit), max = 255))
```

С этой цифрой связана метка 9:

```
train_labels[5]
| [1] 9
```



Рис. 2.2 Пятый образец из нашего набора данных

## 2.2.6 Манипулирование тензорами в R

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*. Давайте посмотрим, какие операции получения среза тензора можно применять с массивами в R.

**ПРИМЕЧАНИЕ** В TensorFlow срез объекта `Tensor` идентичен массиву R, но с некоторыми отличиями. В этом разделе мы сосредоточимся на массивах R, а `Tensor` обсудим в главе 3.

Следующий пример извлекает цифры с 10-й до 99-й и помещает их в массив с формой (90, 28, 28):

```
my_slice <- train_images[10:99, , ]
dim(my_slice)
| [1] 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 14×14 в правом нижнем углу каждого изображения:

```
my_slice <- train_images[, 15:28, 15:28]
dim(my_slice)
| [1] 60000 14 14
```

## 2.2.7 Пакеты данных

В общем случае первая ось во всех тензорах, с которыми вам придется столкнуться в глубоком обучении, будет *осью образцов* (иногда ее называют *измерением образцов*). В примере MNIST «образцы» представляют собой 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[1:128, , ]
```

Вот следующий пакет:

```
batch <- train_images[129:256, , ]
```

А вот  $n$ -й пакет:

```
n <- 3  
batch <- train_images[seq(to = 128 * n, length.out = 128), , ]
```

При рассмотрении таких пакетных тензоров первую ось называют *осью пакета*, или *измерением пакета*. Эти термины часто будут встречаться вам при работе с Keras и другими библиотеками глубокого обучения.

## 2.2.8 Практические примеры тензоров с данными

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

- *векторные данные* – двумерные тензоры с формой (`samples`, `features`), где каждый образец представляет собой вектор числовых атрибутов (признаки);
- *временные ряды, или последовательности* – трехмерные тензоры с формой (`samples`, `timesteps`, `features`), где каждый образец является последовательностью векторов признаков длиной `timesteps`;
- *изображения* – четырехмерные тензоры с формой (`samples`, `height`, `width`, `channels`), где каждый образец – это двумерная сетка пикселей и каждый пиксел представлен вектором значений (каналов);
- *video* – пятимерные тензоры с формой (`samples`, `frames`, `height`, `width`, `channels`), где каждый образец – это последовательность изображений длиной `frames`.

## 2.2.9 Векторные данные

Это наиболее часто встречающаяся форма данных. В таких наборах каждый образец может быть представлен вектором, а пакет, соответственно, представлен 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 указаны возраст, почтовый индекс и доход. Каждый

человек охарактеризован вектором с тремя значениями, соответственно, весь набор данных, описывающий 100 000 человек, можно сохранить в двумерном тензоре с формой  $(100000, 3)$ ;

- набор текстовых документов, где каждый документ представлен количеством повторений каждого слова (из словаря с 20 000 наиболее употребительных слов). Каждый документ можно представить как вектор с 20 000 значений (по одному счетчику на каждое слово из словаря), соответственно, весь набор данных, описывающий 500 документов, можно сохранить в двумерном тензоре с формой  $(500, 20000)$ .

### 2.2.10 Временные ряды, или последовательности данных

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).

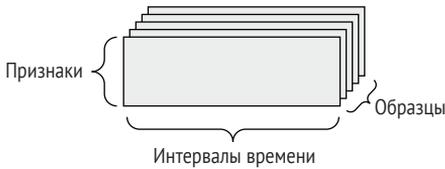


Рис. 2.3 Трехмерный тензор с временным рядом

В соответствии с соглашениями ось времени всегда является второй осью. Рассмотрим несколько примеров.

- Набор данных с котировками акций. Каждую минуту мы сохраняем текущую цену акций, а также наибольшую и наименьшую цены за минувшую минуту. То есть каждая минута представлена одномерным вектором, весь торговый день – двумерным тензором с формой  $(390, 3)$  (где 390 – длительность торгового дня в минутах), а данные за 250 дней – трехмерным тензором с формой  $(250, 390, 3)$ . В данном случае каждый образец представляет данные за один торговый день.
- Набор данных с твитами, где каждый твит кодируется как последовательность 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 глубину цвета. 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).

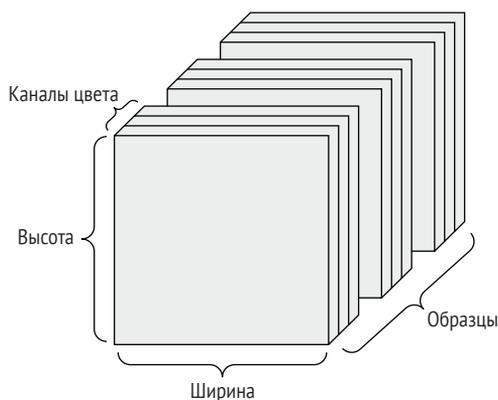


Рис. 2.4 Четырехмерный тензор с изображениями

В отношении форм тензоров с изображениями существует два соглашения: канал цвета следует последним (*channels-last*, используется в TensorFlow) и канал цвета следует первым (*channels-first*, быстро выходит из употребления). Фреймворк машинного обучения TensorFlow, разработанный компанией Google, отводит для цвета последнюю ось:  $(\text{samples}, \text{height}, \text{width}, \text{color\_depth})$ . Во втором типе соглашения для цвета отведена ось, следующая сразу за осью пакетов:  $(\text{samples}, \text{color\_depth}, \text{height}, \text{width})$ . В таком случае предыдущие примеры тензоров будут иметь форму  $(128, 1, 256, 256)$  и  $(128, 3, 256, 256)$ . API Keras поддерживает оба формата.

### 2.2.12 Видеоданные

Видеоданные – один из немногих типов данных, для хранения которых требуются пятимерные тензоры. Видео можно представить как последовательность кадров, где каждый кадр – цветное изображение. Каждый кадр можно сохранить в трехмерном тензоре  $(\text{height}, \text{width}, \text{color\_depth})$ , соответственно, их последовательность можно сохранить в четырехмерном тензоре  $(\text{frames}, \text{height}, \text{width}, \text{color\_depth})$ .

color\_depth), а пакет разных видеороликов – в пятимерном тензоре с формой (samples, frames, height, width, color\_depth).

Например, 60-секундный видеоклип с разрешением 144×256 и частотой 4 кадра в секунду будет состоять из 240 кадров. Для сохранения пакета из четырех таких клипов потребуется тензор с формой (4, 240, 144, 256, 3). То есть 106 168 320 значений! Если предположить, что тензор имеет тип R integer, тогда для хранения каждого значения понадобится 32 бита, то есть для хранения всего тензора – 405 Мбайт. Немало! Видеоролики, с которыми вам придется столкнуться в реальной жизни, намного меньше, потому что они не хранятся как integer и обычно подвергаются значительному сжатию (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 сеть, наложив друг на друга два полносвязных слоя Dense. В библиотеке Keras экземпляр слоя выглядит так:

```
layer_dense(units = 512, activation = "relu")
```

```
| <keras.layers.core.dense.Dense object at 0x7f7b0e8cf520>
```

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

```
output <- relu(dot(W, input) + b)
```

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

- скалярное произведение (dot) исходного тензора input и тензора с именем W;
- сложение (+) получившегося двумерного тензора и вектора b;
- операция relu(x), которая эквивалентна операции max(x, 0).

Даже притом, что в этом разделе очень часто используются выражения из линейной алгебры, вы не найдете здесь математической

нотации. Как нам кажется, программисты без математического образования проще осваивают математические понятия, если они выражены короткими фрагментами кода, а не математическими формулами. Поэтому мы повсюду будем использовать код R и TensorFlow.

### 2.3.1 Поэлементные операции

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

```
naive_relu <- function(x) {
  stopifnot(length(dim(x)) == 2) ← x - двумерный тензор (матрица)
  for (i in 1:nrow(x))
    for (j in 1:ncol(x))
      x[i, j] <- max(x[i, j], 0)
  x
}
```

Точно так же реализуется сложение:

```
naive_add <- function(x, y) {
  stopifnot(length(dim(x)) == 2, dim(x) == dim(y)) ← x и y - двумерные тензоры
  for (i in 1:nrow(x))
    for (j in 1:ncol(x))
      x[i, j] <- x[i, j] + y[i, j]
  x
}
```

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

При работе с массивами R можно пользоваться уже готовыми, оптимизированными встроенными функциями языка R, которые сами делегируют основную работу реализациям базовых подпрограмм линейной алгебры (Basic Linear Algebra Subprograms, BLAS). BLAS – это комплект низкоуровневых, параллельных и эффективных процедур для вычислений с тензорами, которые обычно реализуются на Fortran или C. То есть в R поэлементные операции можно записывать, как показано ниже, и они будут выполняться почти мгновенно:

```
z <- x + y ← Поэлементное сложение
z[z < 0] <- 0 ← Поэлементная relu
```

Давайте сравним время выполнения кода:

```
random_array <- function(dim, min = 0, max = 1)
  array(runif(prod(dim), min, max),
```

```

dim)

x <- random_array(c(20, 100))
y <- random_array(c(20, 100))

system.time({
  for (i in seq_len(1000)) {
    z <- x + y
    z[z < 0] <- 0
  }
})[["elapsed"]]

| [1] 0.009

```

Здесь мы затратили 0.009 с. При этом выполнение стандартной версии занимает 0,72 с:

```

system.time({
  for (i in seq_len(1000)) {
    z <- naive_add(x, y)
    z <- naive_relu(z)
  }
})[["elapsed"]]

| [1] 0.724

```

Точно так же при запуске кода TensorFlow на графическом процессоре поэлементные операции выполняются с помощью полностью векторизованных реализаций CUDA, которые могут наилучшим образом использовать высокопараллельную архитектуру чипа графического процессора.

### 2.3.2 Операции с тензорами разной размерности

Наша предыдущая реализация `naive_add` поддерживает только сложение двумерных тензоров с идентичными формами. Но в `layer_dense()` мы складывали двумерный тензор с вектором. Что происходит при сложении, когда формы складываемых тензоров отличаются?

Нам нужно, чтобы меньший тензор транслировался в соответствии с формой большего тензора (операции с тензорами разной размерности в TensorFlow называются *broadcasting* – трансляция). Трансляция состоит из следующих двух этапов:

- 1 к меньшему тензору добавляют оси, чтобы он совпадал по длине ( $\text{dim}(x)$ ) с большим тензором;
- 2 меньший тензор повторяют вместе с этими новыми осями до совпадения с полной формой большего тензора.

Тензоры TensorFlow, описанные в главе 3, имеют богатые встроенные функции для операций с разной размерностью. Однако в этом

разделе мы осваиваем понятия машинного обучения с нуля, используя массивы R, и намеренно избегаем неявного поведения R при работе с двумя массивами разных размеров. Мы можем реализовать наш собственный подход к трансляции, создав меньший тензор, чтобы он соответствовал форме большего тензора, что возвращает нас к стандартной поэлементной операции.

Рассмотрим конкретный пример. Пусть у нас есть X с формой (32, 10) и y с формой (10):

```
X <- random_array(c(32, 10)) ← X - матрица случайных чисел
                               формы (32, 10)
y <- random_array(c(10)) ← y - случайный вектор формы (10)
```

Сначала добавим одну первую ось к y и получим форму (1, 10):

```
dim(y) <- c(1, 10) ← Теперь у имеет форму (1, 10)
str(y)
```

```
num [1, 1:10] 0.885 0.429 0.737 0.553 0.426 ...
```

Затем повторим y 32 раза вдоль новой оси и получим тензор Y с формой (32, 10), где  $Y[i, ] == y$  для  $i$  в  $\text{seq}(32)$ :

```
Y <- y[rep(1, 32), ] ← Повторим y 32 раза вдоль оси 1
str(Y)                и получим Y с формой (32, 10)
```

```
num [1:32, 1:10] 0.885 0.885 0.885 0.885 0.885 ...
```

Теперь нам осталось лишь сложить X и Y, потому что у них одинаковая форма.

С точки зрения реализации нам бы не хотелось создавать новый тензор ранга 2, потому что это ужасно неэффективно. В большинстве тензорных реализаций, включая R и TensorFlow, операция повторения полностью виртуальна: она происходит на алгоритмическом уровне, а не на физическом уровне памяти. Однако имейте в виду, что соответствующие реализации R и TensorFlow (и NumPy) отличаются по своему поведению (подробности мы рассмотрим в главе 3). Тем не менее представление, что вектор повторяется 10 раз вдоль новой оси, является полезной ментальной моделью. Вот как будет выглядеть буквальная реализация:

```
naive_add_matrix_and_vector <- function(x, y) {
  stopifnot(length(dim(x)) == 2, ← x - двумерный тензор
            length(dim(y)) == 1, ← y - вектор
            ncol(x) == dim(y))
  for (i in seq(dim(x)[1]))
    for (j in seq(dim(x)[2]))
      x[i, j] <- x[i, j] + y[j]
  x
}
```

### 2.3.3 Скалярное произведение тензоров

Скалярное произведение (dot product), также иногда называют *тензорным произведением* (не путайте с поэлементным произведением), – наиболее распространенная и наиболее полезная операция с тензорами. В отличие от поэлементных операций, она объединяет элементы из исходных тензоров. Поэлементное произведение в R выполняется с помощью оператора `*`, а скалярное произведение – с помощью оператора `%**%`:

```
x <- random_array(c(32))
y <- random_array(c(32))
z <- x %**% y
```

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

```
z = x • y
```

Но что делает операция скалярного произведения с точки зрения математики? Для начала разберемся со скалярным произведением двух векторов,  $x$  и  $y$ . Оно вычисляется следующим образом:

```
naive_vector_dot <- function(x, y) {
  stopifnot(length(dim(x)) == 1, | x и y - одномерные векторы
            length(dim(y)) == 1, | одинакового размера
            dim(x) == dim(y))
  z <- 0
  for (i in seq_along(x))
    z <- z + x[i] * y[i]
  z
}
```

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

Также можно получить скалярное произведение матрицы  $x$  на вектор  $y$ , являющееся вектором, элементами которого являются скалярные произведения строк  $x$  на  $y$ . Вот как реализуется эта операция:

```
naive_matrix_vector_dot <- function(x, y) {
  stopifnot(length(dim(x)) == 2, ← x - двумерный тензор (матрица)
            length(dim(y)) == 1, ← y - одномерный тензор (вектор)
            nrow(x) == dim(y)) ←
  z <- array(0, dim = dim(y)) ←
  for (i in 1:nrow(x))
    for (j in 1:ncol(x))
      z[i] <- z[i] + x[i, j] * y[j]
  z
}
```

Первое измерение  $x$  должно иметь одинаковую размерность с  $y$ !

Эта операция возвращает вектор из нулей, совпадающий по форме с  $y$

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

```
naive_matrix_vector_dot <- function(x, y) {
  z <- array(0, dim = c(nrow(x)))
  for (i in 1:nrow(x))
    z[i] <- naive_vector_dot(x[i, ], y)
  z
}
```

Обратите внимание, что если один из двух тензоров имеет больше одного измерения `length(dim(x))`, скалярное произведение `%*%` перестает быть *симметричной* операцией, то есть результат `x %*% y` не совпадает с результатом `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` (`x%*%y`) можно, только если `ncol(x) == nrow(y)`. В результате получится матрица с формой `(nrow(x), ncol(y))`, элементами которой являются скалярные произведения строк `x` на столбцы `y`. Вот как могла бы выглядеть простейшая реализация:

```
naive_matrix_dot <- function(x, y) {
  stopifnot(length(dim(x)) == 2,
            length(dim(y)) == 2,
            ncol(x) == nrow(y))
  z <- array(0, dim = c(nrow(x), ncol(y)))
  for (i in 1:nrow(x)) {
    for (j in 1:ncol(y)) {
      row_x <- x[i, ]
      column_y <- y[, j]
      z[i, j] <- naive_vector_dot(row_x, column_y)
    }
  }
  z
}
```

x и y -  
двумерные  
тензоры  
(матрицы) →

← Первое измерение x должно иметь  
одинаковую размерность с y!

← Эта операция возвращает  
вектор из нулей,  
совпадающий по форме с y

← Итерации по строкам x...

← ...и по столбцам y

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

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

В общем случае скалярное произведение тензоров с большим числом измерений выполняется в соответствии с теми же правилами

совместимости форм, как описывалось выше для случая двумерных матриц:

$$(a, b, c, d) \cdot (d) \rightarrow (a, b, c)$$

$$(a, b, c, d) \cdot (d, e) \rightarrow (a, b, c, e)$$

и т. д.

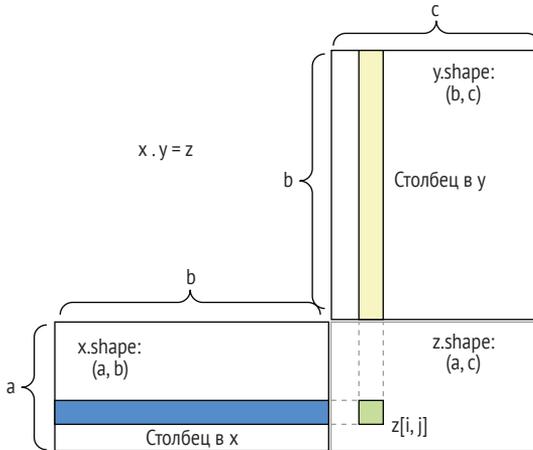


Рис. 2.5 Диаграмма скалярного произведения матриц

### 2.3.4 Изменение формы тензора

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

```
train_images <- array_reshape(train_images, c(60000, 28 * 28))
```

Обратите внимание, что для изменения формы массивов R мы используем функцию `array_reshape()`, а не функцию `dim<-`()`. Это связано с тем, что данные переинтерпретируются с использованием построчной семантики (в отличие от столбцовой семантики R по умолчанию), что, в свою очередь, совместно со способом интерпретации размерности массива числовыми библиотеками, вызываемыми Keras (NumPy, TensorFlow и т. д.). Вы всегда должны использовать функцию `array_reshape()` при изменении массивов R, которые будут переданы в Keras.

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

```
x <- array(1:6)
x
[1] 1 2 3 4 5 6
array_reshape(x, dim = c(3, 2))
  [,1] [,2]
[1,]  1   2
[2,]  3   4
[3,]  5   6
array_reshape(x, dim = c(2, 3))
  [,1] [,2] [,3]
[1,]  1   2   3
[2,]  4   5   6
```

Особый случай изменения формы, который часто встречается в практике, – это *транспозиция*. Транспозиция – это такое преобразование матрицы, когда строки становятся столбцами, а столбцы – строками, то есть  $x[i, ]$  превращается в  $x[, i]$ . Для транспозиции матриц можно использовать функцию `t()`:

```
x <- array(1:6, dim = c(3, 2))
x
  [,1] [,2]
[1,]  1   4
[2,]  2   5
[3,]  3   6
t(x)
  [,1] [,2] [,3]
[1,]  1   2   3
[2,]  4   5   6
```

### 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 = c(0.5, 1)
```

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

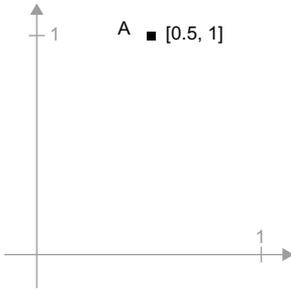


Рис. 2.6 Точка в двумерном пространстве

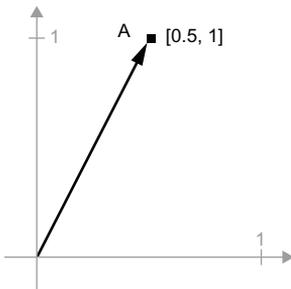


Рис. 2.7 Вектор в двумерном пространстве, изображенный в виде стрелки

Добавим новую точку  $B = c(1, 0.25)$ , построим вектор 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, сложение вектора  $B$  с вектором  $A$  представляет собой действие по копированию точки  $A$  в новое место, расстояние и направление которого от исходной точки  $A$  определяются вектором  $B$ . Если вы сложите вектор с группой точек на плоскости (*объект*), то создадите копию всего объекта в новом месте (см. рис. 2.9). Таким образом, тензорное сложение представляет собой действие по *трансляции* объекта (перемещение объекта без его искажения) на определенную величину в определенном направлении.

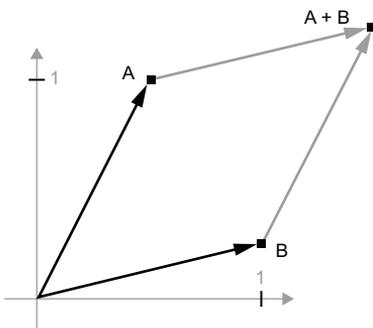


Рис. 2.8 Геометрическая интерпретация суммы двух векторов

В общем случае элементарные геометрические операции, такие как перенос, вращение, масштабирование, наклон и т. д., могут быть выражены как тензорные операции. Вот несколько примеров.

- *Трансляция* – как вы только что видели, добавление вектора к точке сдвинет точку на фиксированную величину в фиксированном направлении. Применительно к набору точек (например, 2D-объекту) это называется трансляцией, или перемещением (см. рис. 2.9).

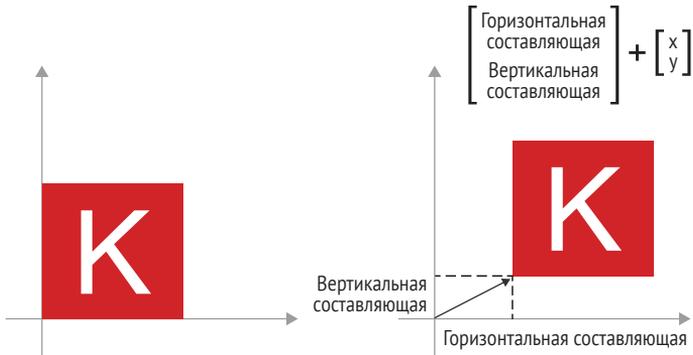


Рис. 2.9 Трансляция в двумерном пространстве при сложении объекта с вектором

- *Вращение* двумерного вектора против часовой стрелки на угол  $\theta$  (рис. 2.10) может быть достигнуто с помощью скалярного произведения с матрицей  $2 \times 2$   $R = \text{rbind}(c(\cos(\theta)), -\sin(\theta)), c(\sin(\theta), \cos(\theta)))$ .

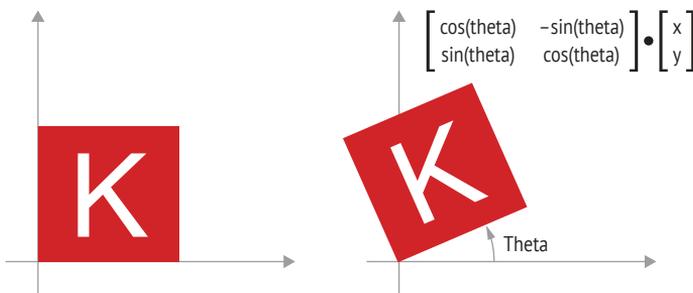


Рис. 2.10 Вращение в двумерном пространстве против часовой стрелки при помощи скалярного произведения

- *Масштабирование* – вертикальное и горизонтальное масштабирование изображения (рис. 2.11) может быть достигнуто с помощью скалярного произведения с матрицей  $2 \times 2$

$S = \text{rbind}(c(\text{horizontal\_factor}, 0), c(0, \text{vertical\_factor}))$  (такая матрица называется *диагональной*, потому что она имеет только ненулевые коэффициенты на своей «диагонали», идущей от верхнего левого угла к нижнему правому).

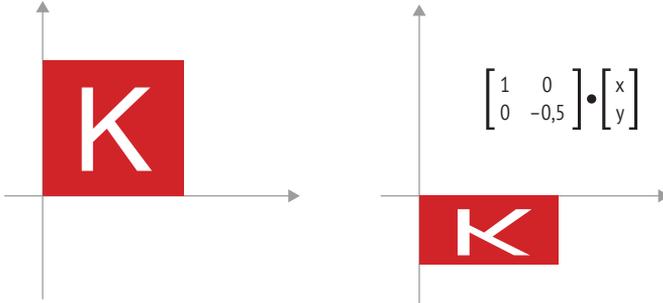


Рис. 2.11 Масштабирование в двумерном пространстве при помощи скалярного произведения

- *Линейное преобразование* – реализуется через скалярное произведение с произвольной матрицей. Заметим, что масштабирование и поворот, перечисленные ранее, по определению являются линейными преобразованиями.
- *Аффинное преобразование* (рис. 2.12) представляет собой комбинацию линейного преобразования (скалярное произведение с матрицей) и трансляции (сложение векторов). Как вы, наверное, уже поняли, именно это вычисление  $y = W \cdot x + b$  реализовано функцией `layer_dense()`! Полносвязный слой Dense без функции активации является аффинным слоем.

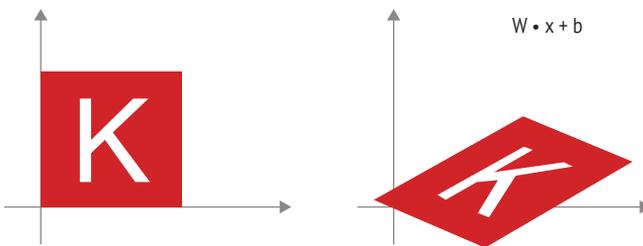


Рис. 2.12 Аффинное преобразование на плоскости

- Полносвязный слой Dense с активацией `relu`. Важная особенность аффинных преобразований заключается в том, что если вы применяете многие из них повторно, вы все равно получите аффинное преобразование. Попробуем проделать это с двумя аффинными преобразованиями:  $\text{affine2}(\text{affine1}(x)) = W_2 \cdot (W_1 \cdot x + b_1) + b_2 = (W_2 \cdot W_1) \cdot x + (W_2 \cdot b_1 + b_2)$ . Это будет аффинное преобразование, где линейная часть – это матрица  $W_2 \cdot W_1$ ,

а трансляционная часть – это вектор  $W_2 \cdot b_1 + b_2$ . Как следствие многослойная нейронная сеть, полностью состоящая из слоев Dense без активаций, будет эквивалентна одному слою Dense. Эта «глубокая» нейронная сеть будет просто замаскированной линейной моделью! Вот почему нам нужны функции активации, такие как `relu` (показана в действии на рис. 2.13). Благодаря функциям активации можно создать цепочку слоев Dense для реализации очень сложных нелинейных геометрических преобразований, что раскрывает очень богатые пространства гипотез для ваших глубоких нейронных сетей. Мы рассмотрим эту идею более подробно в следующей главе.

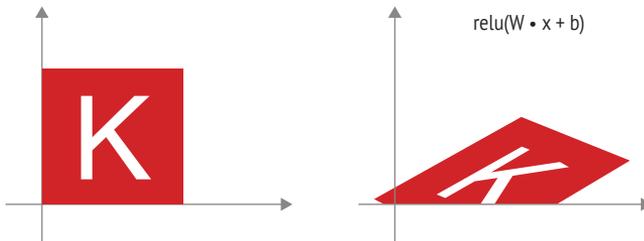


Рис. 2.13 Аффинное преобразование с последующей активацией `relu`

### 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 полезно представить следующую образ в трехмерном пространстве. Вообразите два листа цветной бумаги: один лист красного цвета и другой – синего. Положите их друг на друга. Теперь скомкайте их в маленький комочек. Этот мятый бумажный комочек – 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 (рис. 2.14). 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.

Разглаживание комка бумаги – вот что делает машинное обучение: поиск ясных представлений для сложных перемешанных данных в многомерном пространстве. На данный момент у вас должно сложиться достаточно полное понимание, почему глубокое обуче-

ние преуспевает в этом: оно использует подход последовательного разложения сложных геометрических преобразований в длинную цепь простых, почти так же, как поступает человек, разглаживая комок бумаги. Каждый слой в глубоком обучении применяет преобразование, которое немного распутывает данные, а использование множества слоев позволяет распутывать очень сложные данные.

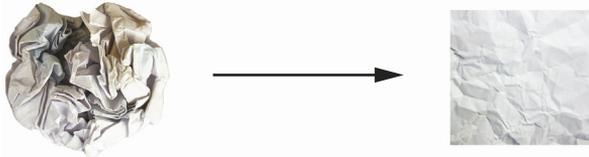


Рис. 2.14 Разглаживание смятого комка исходных данных

## 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 эти весовые матрицы заполняются небольшими случайными значениями (этот шаг называется *случайной инициализацией*). Конечно, бессмысленно было бы ожидать, что  $\text{relu}(\text{dot}(W, \text{input}) + b)$  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. Эта постепенная корректировка, которая также называется *обучением*, составляет суть машинного обучения. Следующие шаги обучения повторяются в *цикле обучения*, пока значение функции потерь не станет достаточно маленьким:

- 1 извлекается пакет обучающих экземпляров  $x$  и соответствующих целей  $y_{\text{true}}$ ;
- 2 нейросеть обрабатывает пакет  $x$  (этот шаг называется *прямым проходом*) и получает пакет предсказаний  $y_{\text{pred}}$ ;
- 3 вычисляются потери сети на пакете, дающие оценку несовпадения между  $y_{\text{pred}}$  и  $y_{\text{true}}$ ;

- 4 корректируются веса сети так, чтобы немного уменьшить потери на этом пакете.

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

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

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

Но такой подход крайне неэффективен, потому что требует выполнять два прямых прохода (что обходится довольно дорого) для каждого отдельного веса (которых очень много, обычно тысячи, а иногда и до нескольких миллионов). К счастью, существует намного более эффективный подход – *градиентный спуск*.

Градиентный спуск – это метод оптимизации, на котором работают современные нейронные сети. Его суть в следующем: все функции, используемые в наших моделях (такие как dot или +), преобразуют входные данные плавным и непрерывным образом. Если взять, к примеру, функцию  $z = x + y$ , небольшое изменение  $y$  приведет к небольшому изменению  $z$ , и если вы знаете направление изменения  $y$ , вы можете сделать вывод о направлении изменения  $z$ . На языке математиков мы говорим, что эта функция *дифференцируема*. Если вы соедините такие функции в цепочку, совокупная функция, которую вы получите, все равно останется дифференцируемой. В частности, это относится к функции, которая сопоставляет коэффициенты модели с потерями модели на пакете данных: небольшое изменение коэффициентов модели приводит к небольшому предсказуемому изменению значения потерь. Это позволяет вам использовать математический оператор, называемый *градиентом*, для описания изменения потерь при перемещении коэффициентов

модели в разных направлениях. Если вы вычисляете этот градиент, вы можете использовать его для перемещения коэффициентов (всех сразу в одном обновлении, а не по одному за раз) в направлении, уменьшающем потери.

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

## 2.4.1 Что такое производная?

Рассмотрим непрерывную гладкую функцию  $f(x) = y$ , отображающую вещественное число  $x$  в новое вещественное число  $y$ . Пример такой функции изображен 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  $y$  изменится на маленькую величину  $\epsilon_y$  (рис. 2.16).

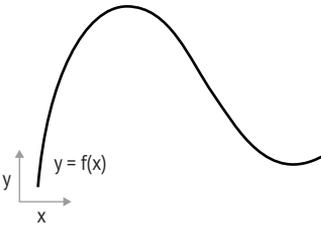


Рис. 2.15 Гладкая функция

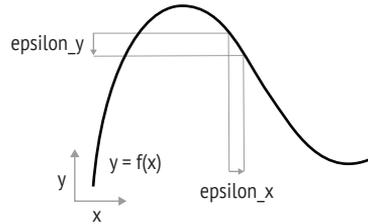


Рис. 2.16 Если функция непрерывна, небольшое изменение  $x$  приводит к небольшому изменению  $y$

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

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

Очевидно, что такая линейная аппроксимация действительна, только когда  $x$  располагается достаточно близко к точке  $p$ . Наклон  $a$  называется *производной*  $f$  в точке  $p$ . Если  $a$  имеет отрицательное значение, значит, небольшое увеличение  $x$  в окрестностях  $p$  приведет к уменьшению  $f(x)$  (как показано на рис. 2.17); а если положительное, то 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 дифференцируемостью подразумевается «имеет производную»: например, глад-

кие непрерывные функции могут иметь производную) существует такая производная функция  $f'(x)$ , которая отображает значения  $x$  в наклон локальной линейной аппроксимации в точках  $f$ . Например, производной от  $\cos(x)$  является  $-\sin(x)$ , производной от  $f(x) = a * x$  является  $f'(x) = a$ , и т. д.



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

Способность дифференцировать функции является очень мощным инструментом, когда дело доходит до *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 только что рассмотрели, превратила скалярное значение  $x$  в другое скалярное значение  $y$ : вы можете построить его как кривую на плоскости. Теперь представьте функцию, которая превращает список скаляров  $(x, y)$  в скалярное значение  $z$ : это будет векторная операция. Вы можете построить его как двумерную поверхность в трехмерном пространстве (индексированном координатами  $x, y, z$ ). Точно так же вы можете представить функции, принимающие в качестве входных данных матрицы, трехмерные тензоры и т. д.

Понятие дифференцирования можно применить к любой такой функции, если поверхности, которые они описывают, непрерывные и гладкие. Производная тензорной операции (или тензорной функции) называется *градиентом*. Градиент – это просто обобщение понятия производной для функций, принимающих тензоры в качестве входных данных. Помните, как для скалярной функции производная представляет локальный наклон кривой функции? Точно так же градиент тензорной функции представляет кривизну многомерной поверхности, описываемой функцией. Он характеризует, как изменяется выход функции при изменении ее входных параметров.

Давайте рассмотрим пример, основанный на машинном обучении. Пусть нам даны:

- входной вектор  $x$  (образец в наборе данных);
- матрица  $W$  (веса модели);
- цель  $y\_true$  (что модель должна научиться связывать с  $x$ );
- функция потерь  $loss\_fn()$  (предназначена для измерения различия между текущими прогнозами модели и  $y\_true$ ).

Вы можете с помощью  $W$  вычислить приближение к цели  $y\_pred$  и определить потери, или несоответствие, между кандидатом  $y\_pred$  и целью  $y\_true$ :

```
y_pred <- dot(W, x)
loss_value <- loss_fn(y_pred, y_true)
```

Мы используем веса модели  $W$ , чтобы получить прогноз  $x$

Вычисляем, насколько сильно отличается прогноз

Теперь пришло время использовать градиенты, чтобы выяснить, как обновить  $W$  для уменьшения  $loss\_value$ . Как это сделать? Если входные данные  $x$  и  $y$  зафиксированы, тогда это можно интерпретировать как функцию, отображающую значения  $W$  в значения потерь:

```
loss_value <- f(W)
```

$f()$  описывает кривую (или многомерную поверхность), образованную значениями потерь при изменении  $W$

Допустим, что  $W_0$  – текущее значение  $W$ . Тогда производной функции  $f$  в точке  $W_0$  будет тензор  $grad(loss\_value, W_0)$  с той же формой, что и  $W$ , в котором каждый элемент  $grad(loss\_value, W_0)[i, j]$  определяет направление и величину изменения в  $loss\_value$ , наблюдаемого при изменении  $W_0[i, j]$ . Тензор  $grad(loss\_value, W_0)$  – это градиент функции  $f(W) = loss\_value$  в  $W_0$ .

### Частные производные

Тензорная операция  $grad(f(W), W)$  (которая принимает в качестве входных данных матрицу  $W$ ) может быть выражена как комбинация скалярных функций  $grad_{ij}(f(W), w_{ij})$ , каждая из которых будет возвращать производную  $loss\_value = f(W)$  относительно коэффициента  $W[i, j]$  функции  $W$ , предполагая, что все остальные коэффициенты постоянны.  $grad_{ij}$  называется частной производной  $f$  по  $W[i, j]$ .

Что конкретно представляет  $grad(loss\_value, W_0)$ ? Выше вы видели, что производную функции  $f(x)$  единственного аргумента можно интерпретировать как наклон кривой  $f$ . Аналогично  $gradient(f)(W_0)$  можно интерпретировать как тензор, описывающий кривизну  $f(W)$  в окрестностях  $W_0$ .

Соответственно, почти так же, как вы можете уменьшить значение  $f(x)$ , немного сдвинув  $x$  в противоположном направлении от

производной, в случае функции  $f(W)$  тензора вы можете уменьшить  $\text{loss\_value} = f(W)$ , переместив  $W$  в направлении, противоположном градиенту: например,  $W1 = W0 - \text{step} * \text{grad}(f(W0), W0)$  (где  $\text{step}$  – небольшой коэффициент масштабирования). Это означает, что нужно идти против направления наибольшего подъема  $f$ , что означает движение в сторону спуска. Обратите внимание, что коэффициент масштабирования  $\text{step}$  необходим, потому что  $\text{grad}(\text{loss\_value}, W0)$  аппроксимирует кривизну только тогда, когда вы близки к  $W0$ , поэтому вам не следует уходить слишком далеко от  $W0$ .

### 2.4.3 Стохастический градиентный спуск

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

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

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

- 1 извлекается пакет обучающих экземпляров  $x$  и соответствующих целей  $y_{\text{true}}$ ;
- 2 сеть обрабатывает пакет  $x$  и получает пакет предсказаний  $y_{\text{pred}}$  (*прямой проход*);
- 3 вычисляются потери сети на пакете, дающие оценку несовпадения между  $y_{\text{pred}}$  и  $y$ ;
- 4 вычисляется градиент потерь для параметров сети (*обратный проход*);
- 5 параметры корректируются на небольшую величину в направлении, противоположном градиенту, например  $W = W - (\text{learning\_gate} * \text{gradient})$ , и тем самым снижаются потери. *Скорость обучения* (здесь  $\text{learning\_gate}$ ) является скалярным компонентом, зависящим от «скорости» градиентного спуска.

Ничего сложного! Я только что описал *стохастический градиентный спуск на небольших пакетах* (mini-batch Stochastic Gradient Descent, minibatch SGD). Термин стохастический отражает тот факт, что каждый пакет данных выбирается случайно (в науке слово *стохастический* считается синонимом слова *случайный*). Рисунок 2.18 иллюстрирует происходящее на примере одномерных данных, когда сеть имеет только один параметр и в вашем распоряжении имеется только один обучающий образец.



Рис. 2.18 Градиентный спуск вниз по одномерной кривой потерь (один обучаемый параметр)

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

Обратите внимание, что алгоритм 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 стохастическим градиентным спуском* (batch SGD). Каждое изменение в этом случае будет более точным, но более дорогостоящим. Эффективным компромиссом между этими двумя крайностями является использование небольших пакетов.

На рис. 2.18 изображен градиентный спуск в одномерном пространстве параметров, но на практике вы будете использовать 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 получить представление о поверхностях потерь, представьте градиентный спуск по двумерной поверхности, как показано на рис. 2.19. Но имейте в виду, что вам не удастся мысленно

представить фактический процесс обучения нейронной сети – нельзя представить 1 000 000-мерное пространство более или менее осмысленным способом. Поэтому всегда помните, что представление, полученное на таких моделях с небольшим числом измерений, может быть не всегда точным на практике. В прошлом это часто приводило к ошибкам исследователей глубокого обучения.

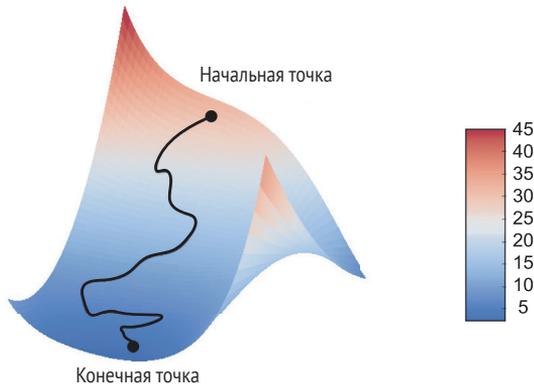


Рис. 2.19 Градиентный спуск вниз по двумерной поверхности потерь (два обучаемых параметра)

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, внимания заслуживает идея *импульса*, которая используется во многих этих вариантах. Импульс вводится для решения двух проблем SGD: невысокой скорости сходимости и попадания в локальный минимум. Взгляните на рис. 2.20, на котором изображена кривая функции потерь для 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 маленьким шагом обучения, тогда процесс оптимизации может застрять в локальном минимуме, не найдя пути к *глобальному минимуму*.

Таких проблем можно избежать, используя 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 также из текущей скорости (набранной в результате действия силы ускорения на предыдущем шаге). На практике это означает, что приращение параметра  $w$  определяется не только по текущему значению градиента, но и по величине предыдущего приращения параметра, как показано в следующей упрощенной реализации:

```

past_velocity <- 0
momentum <- 0.1  ← Постоянный коэффициент импульса
repeat {        ← Цикл оптимизации
  p <- get_current_parameters() ← p включает w, потерю, градиент

  if (p$loss <= 0.01)
    break

  velocity <- past_velocity * momentum + learning_rate * p$gradient
  w <- p$w + momentum * velocity - learning_rate * p$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 мы можем вычислить градиент сложных выражений на практике? Например, как мы можем получить градиент потерь по отношению к весам двухслойной модели, с которой мы начали главу? Вот где вступает в действие *алгоритм обратного распространения*.

### ЦЕПНОЕ ПРАВИЛО

Обратное распространение – это способ использования производных атомарных (элементарных) операций (таких как сложение, `relu`

или тензорное произведение) для простого вычисления градиента произвольно сложных комбинаций этих операций. Важно отметить, что нейронная сеть состоит из множества связанных тензорных операций, каждая из которых имеет простую известную производную. Например, модель, определенная в листинге 2.2, может быть выражена как функция, параметризованная переменными  $W1$ ,  $b1$ ,  $W2$  и  $b2$  (принадлежащими первому и второму слоям Dense соответственно), включающая атомарные операции `dot`, `relu`, `softmax`, и `+`, а также нашу функцию потерь `loss`, которые легко дифференцируются:

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

Формула сообщает нам, что такую цепочку функций можно получить с использованием следующего тождества, которое называется *цепным правилом*.

Рассмотрим две функции  $f$  и  $g$ , а также их комбинацию  $fg$ , такую что  $fg(x) == f(g(x))$ :

```
fg <- function(x) {
  x1 <- g(x)
  y <- f(x1)
  y
}
```

В соответствии с цепным правилом  $grad(y, x) == grad(y, x1) * grad(x1, x)$ . Это позволяет нам легко вычислить производную  $fg$ , если мы знаем производные  $f$  и  $g$ . Цепное правило названо так, потому что, когда вы добавляете больше промежуточных функций, оно начинает выглядеть как цепочка:

```
fghj <- function(x) {
  x1 <- j(x)
  x2 <- h(x1)
  x3 <- g(x2)
  y <- f(x3)
  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, или обратным дифференцированием). Разберем подробнее принцип его работы.

## АВТОМАТИЧЕСКОЕ ДИФФЕРЕНЦИРОВАНИЕ ПРИ ПОМОЩИ ГРАФОВ ВЫЧИСЛЕНИЙ

Удобный способ представления механизма обратного распространения – это графы вычислений. *Граф вычислений* – это структура дан-

ных, лежащая в основе TensorFlow и революции глубокого обучения в целом. Это ориентированный ациклический граф операций – в нашем случае тензорных операций. Например, на рис. 2.21 показано графовое представление нашей первой модели.

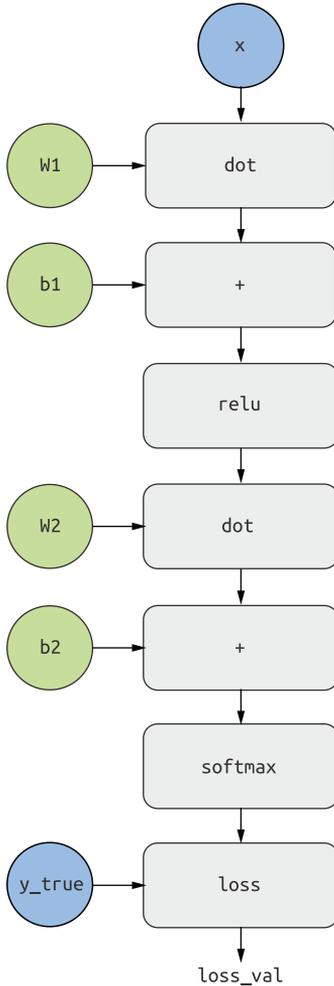


Рис. 2.21 Представление двухслойной модели в виде графа вычислений

Графы вычислений стали чрезвычайно успешной абстракцией в области информатики, потому что они позволяют нам рассматривать вычисления как данные: вычисляемое выражение кодируется как машиночитаемая структура данных, которую можно использовать в качестве входных данных другой программы. Например, можно разработать программу, которая получает граф вычислений и возвращает новый граф вычислений, реализующий крупномасштабную распределенную версию того же вычисления. Это означает, что вы можете распределять любые вычисления без необходимости

самостоятельно писать логику распределения. Или представьте себе программу, которая получает граф вычислений и может автоматически генерировать производную от представляемого им выражения. Это гораздо проще сделать, если ваши вычисления выражены в виде явной графовой структуры данных, а не, скажем, в виде строк символов ASCII в файле .R.

Для лучшего понимания сути обратного распространения рассмотрим очень простой пример графа вычислений (рис. 2.22). Это упрощенная версия рис. 2.21, где у нас есть только один линейный слой и где все переменные являются скалярными. Мы возьмем две скалярные переменные  $w$  и  $b$ , а также скалярный вход  $x$  и применим к ним несколько операций, чтобы объединить их в выход  $y$ . В конце вычислений мы найдем абсолютное значение функции потери:  $\text{loss\_val} = \text{abs}(y\_true - y)$ . Поскольку мы стремимся обновить  $w$  и  $b$  таким образом, чтобы свести к минимуму  $\text{loss\_val}$ , нам нужно вычислить  $\text{grad}(\text{loss\_val}, b)$  и  $\text{grad}(\text{loss\_val}, w)$ .

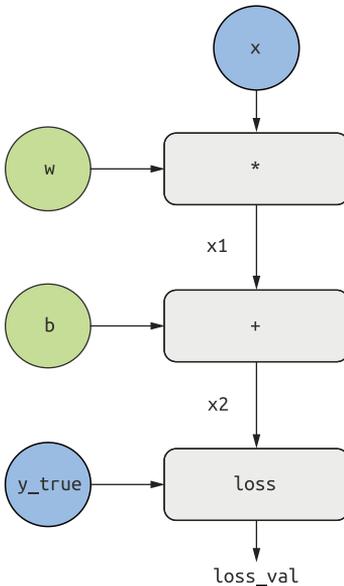


Рис. 2.22 Простой пример графа вычислений

Установим конкретные значения для «входных узлов» графа, то есть вход  $x$ , цель  $y\_true$ ,  $w$  и  $b$ . Мы будем распространять эти значения на все узлы графа сверху вниз, пока не достигнем  $\text{loss\_val}$ . Это *прямой проход* (рис. 2.23).

Теперь давайте «обратим» граф: для каждого ребра в графе, идущего от  $A$  к  $B$ , мы создадим противоположное ребро от  $B$  к  $A$  и спросим, насколько  $B$  изменяется, когда изменяется  $A$ ? То есть что такое  $\text{grad}(B, A)$ ? Пометим каждое перевернутое ребро этим значением. Этот обратный граф представляет *обратный проход* (рис. 2.24).

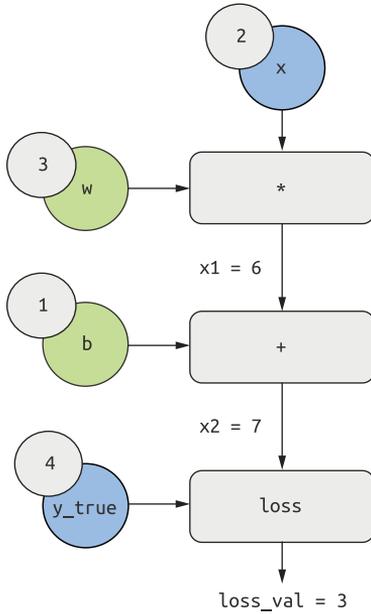


Рис. 2.23 Прямой проход по графу вычислений

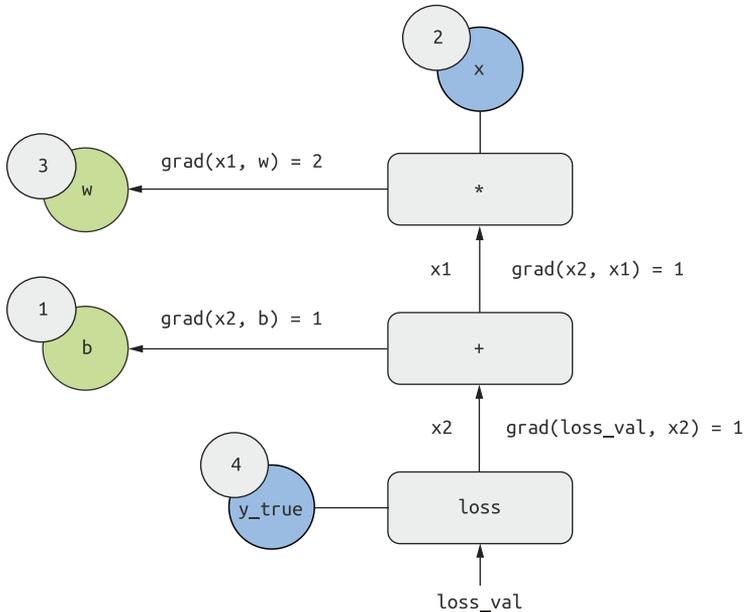


Рис. 2.24 Обратный проход по графу вычислений

Теперь мы знаем следующее:

- $\text{grad}(\text{loss\_val}, x_2) = 1$ , потому что  $\text{loss\_val} = \text{abs}(4 - x_2)$  при изменении  $x_2$  на величину  $\text{epsilon}$  изменяется на ту же величину;

- $\text{grad}(x_2, x_1) = 1$ , потому что  $x_2 = x_1 + b = x_1 + 1$  при изменении  $x_1$  на величину  $\epsilon$  изменяется на ту же величину;
- $\text{grad}(x_2, b) = 1$ , потому что  $x_2 = x_1 + b = 6 + b$  при изменении  $b$  на величину  $\epsilon$  изменяется на ту же величину;
- $\text{grad}(x_1, w) = 2$ , потому что  $x_1 = x * w = 2 * w$  при изменении  $w$  на величину  $\epsilon$  изменяется на  $2 * \epsilon$ .

Из цепного правила для обратного графа следует, что вы можете получить производную узла по отношению к другому узлу, *перемножая производные для каждого ребра на пути, соединяющем два узла*, например  $\text{grad}(\text{loss\_val}, w) = \text{grad}(\text{loss\_val}, x_2) * \text{grad}(x_2, x_1) * \text{grad}(x_1, w)$  (рис. 2.25).

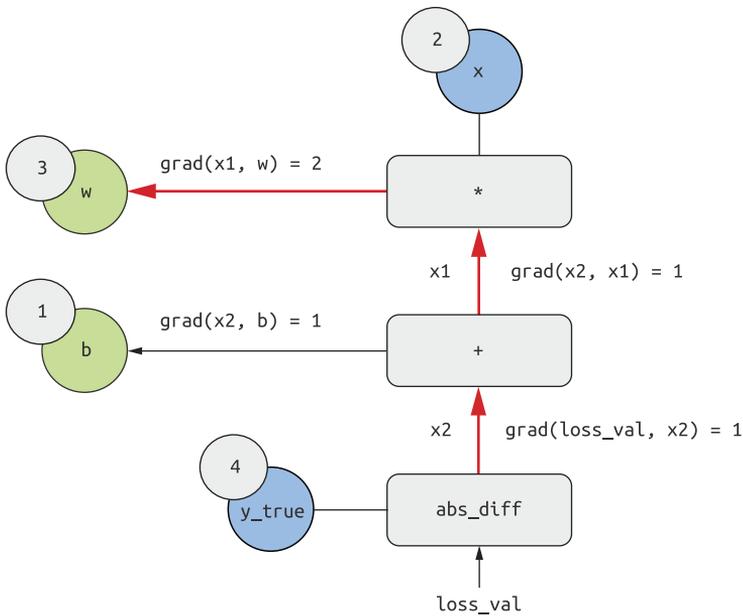


Рис. 2.25 Путь от `loss_val` к `w` в обратном графе

Применяя цепное правило к нашему графу, мы получаем искомые значения:

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

Если в обратном графе есть несколько путей, соединяющих два интересующих нас узла  $a$  и  $b$ , мы получим  $\text{grad}(b, a)$  путем суммирования вкладов всех путей.

Итак, вы только что увидели обратное распространение в действии! Обратное распространение – это просто применение цепного правила к графу вычислений. Больше ничего не нужно. Обратное распространение начинается с окончательного значения потерь и работает в обратном направлении, вычисляя вклад каждого па-

раметра в значение потерь. Отсюда и название «обратное распространение»: мы «распространяем обратно» вклад потерь различных узлов в граф вычислений.

В настоящее время разработчики создают нейронные сети в современных фреймворках, поддерживающих *автоматическое дифференцирование*, таких как TensorFlow. Автоматическое дифференцирование реализовано с помощью графа вычислений, который вы только что видели. Автоматическое дифференцирование позволяет извлекать градиенты произвольных композиций дифференцируемых тензорных операций, не выполняя никакой дополнительной работы, кроме написания прямого прохода. Когда я (Франсуа Шолле) создавал свои первые нейронные сети на C в 2000-х, мне пришлось писать свои градиенты вручную. Теперь, благодаря современным инструментам автоматического дифференцирования, вам никогда не придется самостоятельно реализовывать обратное распространение ошибки. Завидую вашему везению!

## ГРАДИЕНТНАЯ ЛЕНТА В TENSORFLOW

Впечатляющие возможности автоматического дифференцирования TensorFlow доступны нам через API `GradientTape()`. Это диспетчер контекста, который будет «записывать» тензорные операции, выполняемые внутри его области видимости, в виде графа вычислений (иногда называемого *tape* – лента). Затем этот граф можно использовать для получения градиента любого вывода по отношению к любой переменной или набору переменных (экземпляры класса `TensorFlow Variable`). `tf$Variable` – это особый тип тензора, предназначенный для хранения изменяемого состояния. Например, веса нейронной сети всегда являются экземплярами `Variable`:

```

Инициуируем скаляр Variable
начальным значением 0
library(tensorflow)
x <- tf$Variable(0)
with(tf$GradientTape() %as% tape, {
  y <- 2 * x + 3
})
grad_of_y_wrt_x <- tape$gradient(y, x)

Используем «ленту» для получения градиента выхода,
как выхода y при заданном значении переменной x

```

Открываем область видимости GradientTape

Внутри этой области применяем тензорные операции к нашей переменной

Покидаем область видимости

`GradientTape()` выполняет тензорные операции следующим образом:

```

Инициализируем переменную с формой (2, 2)
и начальными нулевыми значениями
x <- tf$Variable(array(0, dim = c(2, 2)))
with(tf$GradientTape() %as% tape, {

```

```

y <- 2 * x + 3
})
grad_of_y_wrt_x <- as.array(tape$gradient(y, x))

```

grad\_of\_y\_wrt\_x – это тензор формы (2, 2) (подобно x), описывающий кривую  $y = 2 * a + 3$  вокруг  $x = \text{array}(0, \text{dim} = \text{c}(2, 2))$

Обратите внимание, что `tape$gradient()` возвращает `Tensor`, который мы конвертируем в массив R с помощью `as.array()`. `GradientTape()` также работает со списками переменных:

```

W <- tf$Variable(random_array(c(2, 2)))
b <- tf$Variable(array(0, dim = c(2)))
x <- random_array(c(2, 2))
with(tf$GradientTape() %as% tape, {
  y <- tf$matmul(x, W) + b
})
grad_of_y_wrt_W_and_b <- tape$gradient(y, list(W, b))
str(grad_of_y_wrt_W_and_b)

```

matmul – это скалярное произведение в TensorFlow

grad\_of\_y\_wrt\_W\_and\_b – это список из двух тензоров с такими же формами, как W и b соответственно

```

List of 2
 $ :<tf.Tensor: shape=(2, 2), dtype=float64, numpy=...>
 $ :<tf.Tensor: shape=(2), dtype=float64, numpy=array([2., 2.]>

```

Мы рассмотрим градиентную ленту более подробно в следующей главе.

## 2.5 Возвращаясь к нашему первому примеру

Вы приближаетесь к концу этой главы, и теперь у вас должно сложиться общее представление о том, что происходит за кулисами в нейронной сети. Волшебный черный ящик, представший перед нами в начале главы, превратился в изящный механизм, изображенный на рис. 2.26: модель, состоящая из связанных слоев, отображает входные данные в прогнозы. Затем функция потерь сравнивает эти прогнозы с целевыми значениями, получая значение потерь: меру того, насколько хорошо прогнозы модели соответствуют ожидаемым. Оптимизатор использует это значение потерь для обновления весов модели.

Давайте вернемся к первому примеру этой главы и рассмотрим каждый его фрагмент в свете новых знаний.

У нас были следующие входные данные:

```

library(keras)
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_images <- array_reshape(train_images, c(60000, 28 * 28))
train_images <- train_images / 255

test_images <- mnist$test$x
test_images <- array_reshape(test_images, c(10000, 28 * 28))

```

```
test_images <- test_images / 255
train_labels <- mnist$train$y
test_labels <- mnist$test$y
```

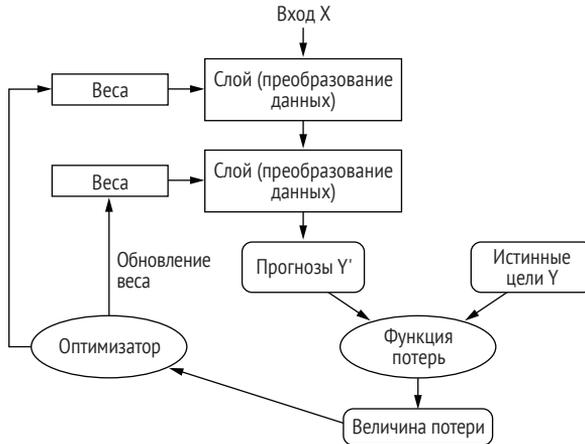


Рис. 2.26 Связь между сетью, уровнями, функцией потерь и оптимизатором

Теперь вы знаете, что входные изображения хранятся в массивах R формы (60000, 784) (обучающие данные) и (10000, 784) (контрольные данные) соответственно.

Так выглядела наша модель:

```
model <- keras_model_sequential(list(
  layer_dense(units = 512, activation = "relu"),
  layer_dense(units = 10, activation = "softmax")
))
```

Теперь вы знаете, что эта модель представляет собой цепочку из двух слоев Dense, что каждый слой применяет несколько простых тензорных операций к входным данным и что в этих операциях участвуют тензоры весов, которые являются атрибутами слоев. Тензоры весов – это то место, где сохраняется *знание* модели.

Дальше у нас следовал этап компиляции модели:

```
compile(model,
  optimizer = "rmsprop",
  loss = "sparse_categorical_crossentropy",
  metrics = c("accuracy"))
```

Теперь вы знаете, что `sparse_categorical_crossentropy` – это функция потерь, применяемая в качестве сигнала обратной связи для изучения тензоров весов и которую фаза обучения пытается минимизировать. Вы также знаете, что уменьшение потерь происходит с помощью мини-пакетного стохастического градиентного спуска.

Точные правила, регулирующие конкретное использование градиентного спуска, определяются оптимизатором `gmsrgor`, переданным в качестве первого аргумента.

Наконец, у нас был цикл обучения модели:

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

Теперь вы должны понимать, что происходит, когда вы вызываете `fit()`: модель начинает перебирать обучающие данные в мини-пакетах по 128 выборок и делает это пять раз (каждая итерация по всем обучающим данным называется *эпохой*). Для каждого пакета модель вычисляет градиент потерь по отношению к весам (используя алгоритм обратного распространения, который выводится из цепного правила) и смещает веса в направлении, которое уменьшит значение потерь для этого пакета.

Пройдя через пять эпох, модель выполнит 2345 градиентных обновлений (469 за эпоху), а потери модели будут достаточно низкими, чтобы она могла классифицировать рукописные цифры с высокой точностью.

На данный момент вы уже знаете большую часть того, что нужно знать о нейронных сетях. Вы можете убедиться в этом, пошагово реализовав упрощенную версию первого примера «с нуля» в TensorFlow.

## 2.5.1 Повторная реализация нашего первого примера с нуля в TensorFlow

Что может лучше продемонстрировать полное и однозначное понимание темы, чем реализация решения задачи с нуля? Конечно, мы углубимся в основы в пределах разумного и не станем переопределять базовые тензорные операции или реализовывать обратное распространение при помощи своего кода. Но мы перейдем на такой низкий уровень, что почти не будем использовать какие-либо функции Keras.

Не волнуйтесь, если в этом примере вам будет не все понятно. В следующей главе мы более подробно рассмотрим API TensorFlow. А пока просто постарайтесь понять суть происходящего. Цель этого примера – закрепить ваше понимание математических основ глубокого обучения с помощью конкретной реализации. Вперед!

### ПРОСТОЙ КЛАСС DENSE

Ранее вы узнали, что слой `Dense` реализует следующее преобразование ввода, где  $W$  и  $b$  – параметры модели, а `activation()` – поэлементная функция, обычно `relu()`, но для последнего слоя это будет `softmax()`:

```
output <- activation(dot(W, input) + b)
```

Давайте реализуем простой слой Dense в виде среды R с атрибутом класса `NaiveDense`, двумя переменными TensorFlow, `W` и `b`, и методом `call()`, который применяет предыдущее преобразование:

```
layer_naive_dense <- function(input_size, output_size, activation) {
  self <- new.env(parent = emptyenv())
  attr(self, "class") <- "NaiveDense"
  self$activation <- activation
  w_shape <- c(input_size, output_size)
  w_initial_value <- random_array(w_shape, min = 0, max = 1e-1)
  self$W <- tf$Variable(w_initial_value)

  b_shape <- c(output_size)
  b_initial_value <- array(0, b_shape)
  self$b <- tf$Variable(b_initial_value)

  self$weights <- list(self$W, self$b)

  self$call <- function(inputs) {
    self$activation(tf$matmul(inputs, self$W) + self$b)
  }

  self
}
```

Создаем матрицу `W` с формой `(input_size, output_size)`, инициализированную случайными значениями

Создаем вектор `b` с формой `(output_size)`, инициализированный нулевыми значениями

Удобный способ получения всех весов слоя

Применяем прямой проход в функции с названием `call`

В этой функции мы придерживаемся операций TensorFlow, чтобы GradientTape мог их отслеживать (вы узнаете больше об операциях TensorFlow в главе 3)

## ПРОСТОЙ КЛАСС SEQUENTIAL

Теперь создадим функцию `naive_model_sequential()` для объединения этих слоев в цепочку, как показано в следующем фрагменте кода. Она оборачивает список слоев и предоставляет метод `call()`, который просто вызывает нижележащие слои по порядку относительно входа. Он также имеет свойство `weights`, чтобы легко отслеживать параметры слоев:

```
naive_model_sequential <- function(layers) {
  self <- new.env(parent = emptyenv())
  attr(self, "class") <- "NaiveSequential"

  self$layers <- layers

  weights <- lapply(layers, function(layer) layer$weights)
  self$weights <- do.call(c, weights)

  self$call <- function(inputs) {
    x <- inputs
    for (layer in self$layers)
      x <- layer$call(x)
    x
  }
}
```

Сглаживает вложенный список до одного уровня

```
self
}
```

Используя классы `NaiveDense` и `NaiveSequential`, мы можем создать черновую модель Keras:

```
model <- naive_model_sequential(list(
  layer_naive_dense(input_size = 28 * 28, output_size = 512,
                    activation = tf$nn$relu),
  layer_naive_dense(input_size = 512, output_size = 10,
                    activation = tf$nn$softmax)
))
stopifnot(length(model$weights) == 4)
```

## ГЕНЕРАТОР ПАКЕТОВ

Далее нам нужен способ перебора данных MNIST в мини-пакетах. Это просто:

```
new_batch_generator <- function(images, labels, batch_size = 128) {
  self <- new.env(parent = emptyenv())
  attr(self, "class") <- "BatchGenerator"

  stopifnot(nrow(images) == nrow(labels))
  self$index <- 1
  self$images <- images
  self$labels <- labels
  self$batch_size <- batch_size
  self$num_batches <- ceiling(nrow(images) / batch_size)

  self$get_next_batch <- function() {
    start <- self$index
    if(start > nrow(images))
      return(NULL) ← Генератор готов

    end <- start + self$batch_size - 1
    if(end > nrow(images))
      end <- nrow(images) ← Последний пакет может быть
                             меньше остальных

    self$index <- end + 1
    indices <- start:end
    list(images = self$images[indices, ],
         labels = self$labels[indices])
  }

  self
}
```

### 2.5.2 Выполнение одного шага обучения

Самая сложная часть процесса – это этап обучения: обновление весов модели после запуска ее на одном пакете данных. Нам нужно сделать следующее:

- 1 вычислить предсказания модели для изображений в пакете;
- 2 вычислить значение потерь для этих прогнозов с учетом фактических меток;
- 3 рассчитайте градиент потерь относительно веса модели;
- 4 переместите веса на небольшую величину в направлении, противоположном градиенту.

Для вычисления градиента мы будем использовать объект TensorFlow GradientTape, который представили в разделе 2.4.4:

```

                                Запуск прямого прохода (вычисление прогнозов
                                модели в области видимости GradientTape
one_training_step <- function(model, images_batch, labels_batch) {
  with(tf$GradientTape() %as% tape, {
    predictions <- model$call(images_batch)
    per_sample_losses <-
      loss_sparse_categorical_crossentropy(labels_batch, predictions)
    average_loss <- mean(per_sample_losses)
  })

```

```

                                Вычисление градиента потерь
                                относительно весов. Выходные градиенты
                                представляют собой список,
                                в котором каждая запись соответствует
                                весу из списка model$weights
                                Обновление весов с использованием градиентов
                                (мы определим эту функцию немного позже)
                                }
  gradients <- tape$gradient(average_loss, model$weights)
  update_weights(gradients, model$weights)
  average_loss
}

```

Как вы уже знаете, цель шага обновления весов, реализованного в функции `update_weights()`, состоит в том, чтобы немного сдвинуть веса в направлении, которое уменьшит потери для текущего пакета. Величина сдвига определяется скоростью обучения – обычно это небольшая величина. Самый простой способ реализовать функцию `update_weights()` – вычесть `gradient * learning_rate` из каждого веса:

```

learning_rate <- 1e-3

update_weights <- function(gradients, weights) {
  stopifnot(length(gradients) == length(weights))
  for (i in seq_along(weights))
    weights[[i]]$assign_sub(
      gradients[[i]] * learning_rate)
}

```

`x$assign_sub(value)` – это эквивалент `x <- x - value` для переменных TensorFlow

На практике вы почти никогда не будете выполнять такой шаг обновления веса вручную. Вместо этого вы должны использовать экземпляр `Optimizer` от Keras:

```

optimizer <- optimizer_sgd(learning_rate = 1e-3)

update_weights <- function(gradients, weights)
  optimizer$apply_gradients(zip_lists(gradients, weights))

```

`zip_lists()` – это вспомогательная функция, которую мы используем для преобразования списков градиентов и весов в список пар вида (градиент, вес). Мы используем ее для сопряжения градиентов с весами для оптимизатора. Например:

```
str(zip_lists(
  gradients = list("grad_for_wt_1", "grad_for_wt_2", "grad_for_wt_3"),
  weights = list("weight_1", "weight_2", "weight_3")))
```

```
List of 3
 $ :List of 2
  ..$ gradients: chr "grad_for_wt_1"
  ..$ weights : chr "weight_1"
 $ :List of 2
  ..$ gradients: chr "grad_for_wt_2"
  ..$ weights : chr "weight_2"
 $ :List of 2
  ..$ gradients: chr "grad_for_wt_3"
  ..$ weights : chr "weight_3"
```

Теперь, когда у нас готов шаг обучения для одиночного пакета, можно переходить к реализации целой эпохи обучения.

### 2.5.3 Полный цикл обучения

Эпоха обучения состоит из повторения шага обучения для каждого пакета обучающих данных, а полный цикл обучения – это просто повторение одной эпохи:

```
fit <- function(model, images, labels, epochs, batch_size = 128) {
  for (epoch_counter in seq_len(epochs)) {
    cat("Epoch ", epoch_counter, "\n")
    batch_generator <- new_batch_generator(images, labels)
    for (batch_counter in seq_len(batch_generator$num_batches)) {
      batch <- batch_generator$get_next_batch()
      loss <- one_training_step(model, batch$images, batch$labels)
      if (batch_counter %% 100 == 0)
        cat(sprintf("loss at batch %s: %.2f\n", batch_counter, loss))
    }
  }
}
```

Давайте испытаем наш код в действии:

```
mnist <- dataset_mnist()
train_images <- array_reshape(mnist$train$x, c(60000, 28 * 28)) / 255
test_images <- array_reshape(mnist$test$x, c(10000, 28 * 28)) / 255
test_labels <- mnist$test$y
train_labels <- mnist$train$y

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

```

Epoch 1
loss at batch 100: 2.37
loss at batch 200: 2.21
loss at batch 300: 2.15
loss at batch 400: 2.09
Epoch 2
loss at batch 100: 1.98
loss at batch 200: 1.83
loss at batch 300: 1.83
loss at batch 400: 1.75
...
Epoch 9
loss at batch 100: 0.85
loss at batch 200: 0.68
loss at batch 300: 0.83
loss at batch 400: 0.76
Epoch 10
loss at batch 100: 0.80
loss at batch 200: 0.63
loss at batch 300: 0.78
loss at batch 400: 0.72

```

### 2.5.4 Оценка модели

Мы можем оценить модель, взяв `max.col()` ее прогнозов по тестовым изображениям и сравнив с ожидаемыми метками:

```

max.col(x) представляет собой векторную
реализацию apply(x, 1, which.max))
predictions <- model$call(test_images)
predictions <- as.array(predictions)
predicted_labels <- max.col(predictions) - 1
matches <- predicted_labels == test_labels
cat(sprintf("accuracy: %.2f\n", mean(matches)))
accuracy: 0.82

```

Преобразуем объект Tensor  
TensorFlow в массив R

Вычитаем 1, потому  
что позиции смещены  
относительно меток на 1,  
например первая позиция  
соответствует цифре 0

Готово! Как видите, сделать «вручную» работу, которую выполняют несколько строк кода Keras, не так уж сложно. Но благодаря тому, что вы прошли эти шаги, теперь у вас должно быть кристально ясное понимание того, что происходит внутри нейронной сети, когда вы вызываете функцию `fit()`. Наличие простой ментальной модели того, что происходит за кулисами вашего кода, позволит вам лучше использовать высокоуровневые функции Keras API.

## Краткие итоги главы

- Тензоры лежат в основе современных систем машинного обучения. Они различаются по типу, рангу и форме.
- Вы можете применять к числовым тензорам различные тензорные операции (такие как сложение, тензорное произведение или поэлементное умножение), которые можно интерпретировать как кодирование геометрических преобразований. Вообще говоря, в глубоком обучении все поддается геометрической интерпретации.
- Модели глубокого обучения состоят из цепочек простых тензорных операций, параметризованных весами, которые сами по себе являются тензорами. Веса модели – это место, где хранятся ее «знания».
- Обучение означает поиск набора весов модели, который минимизирует функцию потерь для заданного набора образцов обучающих данных и соответствующих им целей.
- Обучение происходит путем создания случайных пакетов выборок данных и их целей и вычисления градиента параметров модели по отношению к потерям в пакете. Затем параметры модели немного перемещаются (величина перемещения определяется скоростью обучения) в направлении, противоположном градиенту. Это называется мини-пакетным стохастическим градиентным спуском.
- В целом процесс обучения возможен благодаря тому, что все тензорные операции в нейронных сетях дифференцируемы, и, таким образом, можно применить цепное правило вывода, чтобы найти функцию градиента, отображающую текущие параметры и текущий пакет данных в значение градиента. Это называется обратным распространением ошибки.
- Два ключевых понятия, которые вы будете часто встречать в следующих главах, – это *потеря* и *оптимизатор*. Необходимо дать им четкое определение, прежде чем вы начнете вводить данные в модель:
  - *потеря* – это величина, которую вы пытаетесь минимизировать во время обучения, поэтому она представляет собой меру успеха задачи, которую вы пытаетесь решить;
  - *оптимизатор* определяет точный способ использования градиента потерь для обновления параметров: например, это может быть оптимизатор RMSprop, SGD с импульсом и т. д.

# Введение в Keras и TensorFlow

---

## **Эта глава охватывает следующие темы:**

- более детальный рассказ о TensorFlow, Keras и их взаимосвязи;
- настройка среды разработки для глубокого обучения;
- перенос концепций глубокого обучения в Keras и TensorFlow.

Эта глава содержит все необходимое, чтобы начать применять глубокое обучение на практике. Я кратко расскажу вам о Keras (<https://keras.rstudio.com>) и TensorFlow (<https://tensorflow.rstudio.com>) – инструментах глубокого обучения на основе R, которые мы будем использовать на протяжении всей книги. Вы узнаете, как настроить среду разработки для глубокого обучения с поддержкой TensorFlow, Keras и GPU. Наконец, опираясь на знания о Keras и TensorFlow, полученные в главе 2, мы рассмотрим основные компоненты нейронных сетей и их представление в API Keras и TensorFlow.

Завершив чтение этой главы, вы сможете перейти к изучению практических примеров, которые начинаются с главы 4.

## 3.1 Что такое TensorFlow?

TensorFlow – это бесплатная платформа машинного обучения с открытым исходным кодом, разработанная в первую очередь Google. Как и в самом R, основная цель TensorFlow – дать ученым, инженерам и исследователям возможность манипулировать математическими выражениями с помощью числовых тензоров. Но TensorFlow привносит в R следующие новые возможности:

- он может автоматически вычислять градиент любого дифференцируемого выражения (как вы видели в главе 2), что делает его идеально подходящим для машинного обучения;
- он может работать не только на процессорах, но и на GPU и TPU, которые являются высокопараллельными аппаратными ускорителями;
- графовые вычисления, определенные в TensorFlow, можно легко распределить между многими машинами;
- программы TensorFlow можно экспортировать в другие среды выполнения, такие как C++, JavaScript (для браузерных приложений) или TensorFlow Lite (для приложений, работающих на мобильных или встроенных устройствах). Это упрощает развертывание приложений TensorFlow на различных устройствах.

Стоит отметить, что TensorFlow – это гораздо больше, чем библиотека. Это полноценная платформа, на которой построена обширная экосистема компонентов. Некоторые из них разработаны Google, а остальные – сторонними компаниями. Например, есть TF-Agents для исследований в области обучения с подкреплением, TFX для эффективного управления рабочим процессом машинного обучения, TensorFlow Serving для производственного развертывания и репозиторий предварительно обученных моделей TensorFlow Hub. Вместе эти компоненты охватывают очень широкий спектр вариантов применения, от передовых научных исследований до крупномасштабных производственных приложений.

TensorFlow очень хорошо масштабируется: например, ученые из Окриджской национальной лаборатории использовали его для обучения модели прогнозирования экстремальных погодных условий с производительностью 1,1 экзафлопс на 27 000 графических процессорах суперкомпьютера IBM Summit. Аналогичным образом в Google использовали TensorFlow для разработки очень ресурсоемких приложений глубокого обучения, таких как играющая в шахматы и го нейросеть AlphaZero. Если вы располагаете достаточным бюджетом, то можете рассчитывать на масштабирование вычислительных ресурсов для своих моделей примерно до 10 петафлопс на модулях TPU или большом кластере графических процессоров, арендованных в Google Cloud или AWS.

## 3.2 Что такое Keras?

Keras – это высокоуровневый API глубокого обучения на основе TensorFlow, который предоставляет удобный способ определения и обучения любой модели глубокого обучения. Изначально Keras разрабатывали для проведения быстрых экспериментов с глубоким обучением.

Благодаря TensorFlow Keras может работать на различных типах оборудования – графическом процессоре, TPU или обычном процессоре – и легко масштабируется до тысяч машин (рис. 3.1).

Keras славится тем, что создан для разработчиков. Это API для людей, а не для машин. Он реализует передовые методы снижения когнитивной нагрузки: предлагает простые последовательные рабочие процессы, сводит к минимуму количество действий в типовых сценариях использования и обеспечивает четкую и действенную обратную связь в случае ошибки пользователя. Это делает Keras простым в освоении для новичков и очень продуктивным в использовании для экспертов.



Рис. 3.1 Keras и TensorFlow: TensorFlow – это низкоуровневая платформа тензорных вычислений, а Keras – высокоуровневый API глубокого обучения

По состоянию на конец 2021 года у Keras более миллиона пользователей, от академических ученых, инженеров и специалистов по данным в стартапах и крупных компаниях до аспирантов и любителей. Keras активно используют в Google, Netflix, Uber, CERN, NASA, Yelp, Instacart, Square и сотнях стартапов, работающих во всех отраслях. Ваши рекомендации на YouTube созданы моделями Keras. Беспилотные автомобили Waymo разработаны с использованием моделей Keras. На всемирно известном веб-сайте соревнований по машинному обучению Kaggle большинство конкурсов среди разработчиков моделей глубокого обучения было выиграно с использованием Keras.

Благодаря обширной и разнообразной пользовательской базе Keras не заставляет вас следовать единственному «верному» способу построения и обучения моделей. Напротив, он обеспечивает широкий спектр различных рабочих процессов, от очень высокого

уровня до очень низкого, соответствующих различным профилям пользователей. Например, у вас есть несколько способов построения и обучения моделей, каждый из которых представляет собой определенный компромисс между удобством использования и гибкостью. В главе 5 мы подробно рассмотрим значительную часть этого спектра рабочих процессов. Вы можете использовать Keras так же, как и большинство других высокоуровневых фреймворков, – просто вызывая метод `fit()` и позволяя фреймворку делать свое дело – или можете использовать его так же, как базовый R, полностью контролируя каждую мелочь.

Следовательно, все, что вы изучите сейчас, останется актуальным, когда вы станете экспертом. Вы можете легко начать работу, а затем постепенно погрузиться в более сложные рабочие процессы, где вы пишете все больше и больше логики с нуля. Вам не придется переключаться на совершенно другой фреймворк, когда вы из студента превращаетесь в исследователя или из специалиста по данным в разработчика моделей глубокого обучения.

Кстати, это очень похоже на философию языка R! Некоторые языки предлагают только один способ написания программ – например, объектно-ориентированное программирование или функциональное программирование. В отличие от них, R является языком с гибкой парадигмой: он предлагает множество способов использования, которые прекрасно работают вместе. Поэтому R успешно применяется в разных отраслях: наука о данных, машинное обучение, веб-разработка... или просто обучение программированию. В каком-то смысле Keras можно рассматривать как R для глубокого обучения: удобный язык глубокого обучения, который предлагает множество готовых рабочих процессов для разных пользователей.

### 3.3 *Keras и TensorFlow: краткая история*

Keras появился раньше TensorFlow на восемь месяцев. Он был выпущен в марте 2015 года, а TensorFlow – в ноябре 2015 года. Вы можете спросить, если Keras – это надстройка над TensorFlow, как он мог появиться раньше TensorFlow? Первоначально Keras был построен на основе Theano, еще одной библиотеки для работы с тензорами, которая обеспечивала автоматическое дифференцирование и поддержку графического процессора – самую раннюю в своем роде. Фреймворк Theano, разработанный в Монреальском институте алгоритмов обучения (MILA) Университета Монреаля, во многом был предшественником TensorFlow. Он впервые предложил использовать статические графы вычислений для автоматического дифференцирования и для компиляции кода как для обычных, так и для графических процессоров.

В конце 2015 года, после выпуска TensorFlow, Keras был преобразован в API с многовариантной поддержкой: стало возможным использовать Keras либо с Theano, либо с TensorFlow, а переключение между ними было столь же простым, как изменение переменной среды. К сентябрю 2016 года разработка стабильной версии TensorFlow была в целом завершена, и стало возможным сделать его серверной частью по умолчанию для Keras. В 2017 году в перечень поддерживаемых платформ Keras были добавлены две новые опции: CNTK (разработан Microsoft) и MXNet (разработан Amazon). В настоящее время и Theano, и CNTK не разрабатываются, а MXNet не используется широко за пределами Amazon. Keras снова стал монолитным серверным API – поверх TensorFlow.

Keras и TensorFlow уже много лет поддерживают тесную взаимовыгодную связь. В течение 2016 и 2017 годов Keras приобрел популярность как удобный способ разработки приложений, привлекающий новых пользователей в экосистему TensorFlow. К концу 2017 года большинство пользователей TensorFlow использовали его только через Keras или в сочетании с Keras. В 2018 году руководители проекта TensorFlow выбрали Keras в качестве официального высокоуровневого API TensorFlow. В результате API Keras занимает центральное место в TensorFlow 2.0, выпущенном в сентябре 2019 года, – обширном переработанном релизе TensorFlow и Keras, учитывающем отзывы пользователей и опыт практического использования более четырех лет.

### 3.4 Интерфейсы Python и R: краткая история

Интерфейсы R для TensorFlow и Keras стали доступны в конце 2016 и начале 2017 года соответственно. В основном их разрабатывает и поддерживает RStudio.

Интерфейсы R для Keras и TensorFlow построены на основе пакета `reticulate`, который встраивает полный процесс Python в R. Для большинства пользователей это просто одна из деталей реализации. Однако по мере вашего профессионального развития эта особенность архитектуры окажется большим подспорьем, потому что она открывает полный доступ ко всем возможностям как в Python, так и в R.

На протяжении всей книги мы используем интерфейс R для Keras, который хорошо работает с идиомами R. Однако в главе 13 мы покажем, как можно напрямую использовать библиотеку Python из R, даже если для нее нет удобного интерфейса R.

Если вы внимательно прочитали предыдущие главы, то сейчас готовы перейти к применению кода Keras и TensorFlow на практике. Давайте начнем!

## 3.5 Настройка среды разработки для глубокого обучения

Прежде чем браться за разработку приложений для глубокого обучения, вам необходимо настроить среду разработки. Настоятельно рекомендуется, хотя и не обязательно, запускать код глубокого обучения на современном графическом процессоре NVIDIA, а не на процессоре вашего компьютера. Некоторые приложения – в частности, обработка изображений с помощью сверточных сетей – будут мучительно медленно работать даже на быстром многоядерном процессоре. И даже если приложение можно запустить на обычном процессоре, при использовании современного графического процессора скорость выполнения возрастет в 5–10 раз.

Чтобы запускать приложения глубокого обучения на графическом процессоре, можете воспользоваться одним из трех вариантов:

- купите и установите физический графический процессор NVIDIA на свою рабочую машину;
- воспользуйтесь арендованными экземплярами GPU в Google Cloud или Amazon EC2;
- воспользуйтесь бесплатной средой выполнения GPU от Kaggle, Colaboratory или аналогичных поставщиков.

Бесплатные онлайн-провайдеры, такие как Colaboratory или Kaggle, – это самый простой способ начать работу, потому что они позволяют обойтись без покупки оборудования и установки программного обеспечения. Просто откройте вкладку в браузере и начните программировать. Однако бесплатная версия этих сервисов подходит только для небольших рабочих нагрузок. Если вы хотите увеличить масштаб, вам придется использовать первый или второй вариант.

Если у вас еще нет графического процессора, который можно использовать для глубокого обучения (последний высокопроизводительный графический процессор NVIDIA), то эксперименты по глубокому обучению в облаке – это простой и недорогой способ перейти к более серьезным задачам без покупки дополнительного оборудования.

Однако если вы интенсивно занимаетесь глубоким обучением, этот вариант будет невыгоден для вас в долгосрочной перспективе – порой даже в течение нескольких месяцев. Облачные экземпляры недешевы: в середине 2021 года они стоили 2,48 долл. в час за GPU V100 в Google Cloud. Между тем надежный графический процессор потребительского класса будет стоить примерно от 1500 до 2500 долларов – цена, которая остается довольно стабильной, даже несмотря на постоянное улучшение характеристик графических процессоров. Если вы планируете активно заниматься глубоким обучением более

полугода, стоит подумать о покупке локального компьютера с одним или несколькими графическими процессорами.

Кроме того, независимо от того, работаете ли вы локально или в облаке, лучше использовать рабочую станцию Unix. Хотя технически возможно напрямую запустить Keras в среде Windows, мы не рекомендуем это делать. Если вы являетесь убежденным пользователем Windows и хотите запускать задачи глубокого обучения на своем компьютере, самое простое решение – настроить двойную загрузку Windows/Ubuntu на вашем компьютере или использовать подсистему Windows для Linux (WSL), которая позволяет запускать приложения Linux из Windows. Это может показаться хлопотным, но в конечном итоге сэкономит вам много времени и избавит от проблем.

### 3.5.1 Установка Keras и TensorFlow

Установить Keras и TensorFlow на вашем локальном компьютере очень просто:

- 1 убедитесь, что у вас установлен R. Самые свежие инструкции по установке всегда доступны на сайте <https://cloud.r-project.org>;
- 2 установите среду разработки RStudio, доступную для скачивания по адресу <http://mng.bz/v6JM> (вы можете смело пропустить этот шаг, если предпочитаете использовать R из другой среды);
- 3 в консоли R выполните следующие команды:

```
install.packages("keras")
library(reticulate)
virtualenv_create("r-reticulate", python = install_python())

library(keras)
install_keras(envname = "r-reticulate")
```

← Также будут загружены все зависимости R, такие как reticulate

← Настройка R (reticulate) для взаимодействия с Python

← Установка TensorFlow и Keras (модули Python)

Вот и все! Теперь у вас установлены Keras и TensorFlow.

### УСТАНОВКА CUDA

Это важно: если на вашем компьютере есть графический процессор NVIDIA и вы хотите, чтобы TensorFlow использовал его, вам также следует загрузить и установить драйверы CUDA, cuDNN и графического процессора, которые доступны для загрузки по адресам <https://developer.nvidia.com/cuda-downloads> и <https://developer.nvidia.com/cudnn>.

Для каждой версии TensorFlow требуется определенная версия CUDA и cuDNN, и крайне редко последняя версия CUDA работает с последней версией TensorFlow. Как правило, вам нужно будет определить конкретную версию CUDA, необходимую для TensorFlow,

а затем установить ее из архива набора инструментов CUDA по адресу <https://developer.nvidia.com/cuda-toolkit-archive>.

Вы можете узнать, какая версия CUDA необходима для текущей версии TensorFlow, на странице <http://mng.bz/44pV>. Если вы используете более старую версию TensorFlow, ознакомьтесь с таблицей «Проверенные конфигурации» на странице <https://www.tensorflow.org/install/source#gpu>, чтобы найти запись, соответствующую вашей версии TensorFlow.

Вы можете узнать версию TensorFlow, установленную на вашем компьютере, с помощью команды

```
tensorflow::tf_config()
```

```
TensorFlow v2.8.0
```

```
➔ (~/.virtualenvs/r-reticulate/lib/python3.9/site-packages/tensorflow)
Python v3.9 (~/.virtualenvs/r-reticulate/bin/python)
```

На момент написания этой книги для последней версии TensorFlow 2.8 требовались CUDA 11.2 и cuDNN 8.1.

Обратите внимание, что групповые команды (так называемые «заклинания»), которые вы можете запустить на терминале для установки всех драйверов CUDA, очень быстро устаревают (не говоря уже о конкретных командах для каждой версии ОС). Мы не включаем такие команды в книгу, потому что они, скорее всего, устареют еще до того, как книга будет напечатана. Вместо этого мы рекомендуем найти последние инструкции по адресу <https://tensorflow.rstudio.com/installation/>.

Теперь вы можете запустить код Keras. Далее давайте посмотрим, как ключевые идеи, о которых вы узнали в главе 2, превращаются в код Keras и TensorFlow.

## 3.6 Первые шаги с TensorFlow

Как было сказано в предыдущих главах, обучение нейронной сети опирается на два базовых понятия:

- во-первых, низкоуровневые манипуляции с тензорами – инфраструктура, лежащая в основе всего современного машинного обучения. Вот за что отвечает API TensorFlow:
  - *тензоры*, в том числе специальные тензоры, хранящие состояние сети (переменные);
  - *тензорные операции*, такие как сложение, `relu`, `matmul`;
  - *обратное распространение* – способ вычисления градиента математических выражений (обрабатывается в TensorFlow через объект `GradientTape`);
- во-вторых, компоненты глубокого обучения высокого уровня. За них отвечает API Keras:

- *слои*, которые объединяются в модель;
- *функция потерь*, которая определяет сигнал обратной связи, используемый для обучения;
- *оптимизатор*, который определяет, как происходит обучение;
- *метрики* для оценки качества модели, такие как точность;
- *обучающий цикл*, выполняющий мини-пакетный стохастический градиентный спуск.

В предыдущей главе мы уже кратко рассмотрели некоторые компоненты API TensorFlow и Keras. Мы использовали класс `Variable` TensorFlow, операцию `matmul` и `GradientTape`, создали полносвязные слои Keras, упаковали их в последовательную модель и обучили эту модель с помощью метода `fit()`.

Далее разберем практическую реализацию этих шагов с помощью TensorFlow и Keras.

### 3.6.1 Тензоры TensorFlow

Чтобы что-то делать в TensorFlow, нам понадобятся тензоры. В предыдущей главе вы познакомились с понятием тензора, его свойствами и соответствующей терминологией, но в качестве примера реализации использовали массивы R. В этой главе мы представляем практическую реализацию тензоров, используемых TensorFlow.

Тензоры TensorFlow очень похожи на массивы R; они являются контейнером для данных, которые также имеют некоторые метаданные, такие как форма и тип. Вы можете преобразовать массив R в тензор TensorFlow с помощью `as_tensor()`:

```
r_array <- array(1:6, c(2, 3))
tf_tensor <- as_tensor(r_array)
tf_tensor

tf.Tensor(
[[1 3 5]
 [2 4 6]], shape=(2, 3), dtype=int32)
```

Как и массивы R, тензоры поддерживают многие уже знакомые вам тензорные операции – функции, такие как `dim()`, `length()`; встроенные математические вычисления, такие как `+` и `log()`, и т. д.:

```
dim(tf_tensor)

[1] 2 3

tf_tensor + tf_tensor

tf.Tensor(
[[ 2 6 10]
 [ 4 8 12]], shape=(2, 3), dtype=int32)
```

Набор базовых функций R для работы с тензорами весьма обширен:

```
methods(class = "tensorflow.tensor")
[1] -      !      !=     [      [<-    *
[7] /      &      %/%   %%     ^      +
[13] <      <=     ==     >      >=    |
[19] abs    acos    all    any    aperm  Arg
[25] asin   atan    cbind  ceiling Conj  cos
[31] cospi  digamma dim    exp    expm1  floor
[37] Im     is.finite is.infinite is.nan length lgamma
[43] log    log10   log1p  log2   max    mean
[49] min    Mod     print  prod   range  rbind
[55] Re     rep     round  sign   sin    sinpi
[61] sort   sqrt    str    sum    t      tan
[67] tanpi
```

Это означает, что в большинстве случаев вы можете использовать для тензоров TensorFlow и для массивов R один и тот же код.

### 3.7 Атрибуты тензоров

В отличие от массивов R, тензоры имеют некоторые атрибуты, к которым вы можете получить доступ с помощью оператора `$`:

```
tf_tensor$ndim ← ndim возвращает скалярное целое число,
[1] 2              ранг тензора, эквивалентный length(dim(x))
```

Векторы R длины 1 автоматически преобразуются в тензоры ранга 0, тогда как векторы R длины  $> 1$  преобразуются в тензоры ранга 1:

```
as_tensor(1)$ndim
[1] 0
as_tensor(1:2)$ndim
[1] 1
tf_tensor$shape
TensorShape([2, 3])
```

`tf_tensor$shape` возвращает объект `tf.TensorShape`. Это объект класса с поддержкой неопределенных или неуказанных размерностей, а также различных методов и свойств:

```
methods(class = class(shape())[1])
[1] !=      [      [[      [[<-    [<-    ==
```

```
[7] as_tensor as.double as.integer as.list as.numeric c
[13] format length merge print
```

На данный момент вам достаточно знать, что вы можете преобразовать `TensorShape` в целочисленный вектор с помощью `as.integer()` (`dim(x)` – это сокращение от `as.integer(x$shape)`), а также можете создать объект `TensorShape` вручную с помощью функции `shape()`:

```
shape(2, 3)

TensorShape([2, 3])

tf_tensor$type

tf.int32
```

`tf_tensor$type` возвращает тип данных массива. TensorFlow поддерживает намного более широкий спектр типов данных, чем базовый R. Например, базовый R имеет один целочисленный тип `integer`, тогда как TensorFlow поддерживает целых 13! Целочисленный тип R соответствует `int32`. Различные типы данных требуют различных компромиссов между занимаемым объемом памяти и диапазоном значений, которые они могут представлять. Например, тензор с типом данных `int8` занимает в четыре раза меньше места в памяти по сравнению с тензором `int32`, но он может представлять только целые числа от  $-128$  до  $127$ , а не от  $-2\,147\,483\,648$  до  $2\,147\,483\,647$ .

На протяжении всей книги мы также будем иметь дело с данными с плавающей запятой. Тип данных с плавающей запятой R (по умолчанию `double`) преобразуется в TensorFlow в `tf.float64`:

```
r_array <- array(1)
typeof(r_array)

[1] "double"

as_tensor(r_array)$dtype

tf.float64
```

В этой книге мы будем в основном использовать `float32` в качестве типа данных с плавающей запятой по умолчанию, жертвуя некоторой точностью ради меньшего объема занимаемой памяти и более высокой скорости вычислений:

```
as_tensor(r_array, dtype = "float32")

tf.Tensor([1.], shape=(1), dtype=float32)
```

### 3.7.1 Форма тензора и ее изменение

Функция `as_tensor()` принимает дополнительный аргумент `shape` (форма), который можно использовать для расширения скаляра или

изменения формы тензора. Например, чтобы создать массив нулей, вы можете написать:

```
as_tensor(0, shape = c(2, 3))

tf.Tensor(
[[0. 0. 0.]
 [0. 0. 0.]], shape=(2, 3), dtype=float32)
```

Вы также можете изменить форму тензора для векторов R, которые не являются скалярами ( $\text{length}(x) > 1$ ), если общий размер массива остается прежним:

```
as_tensor(1:6, shape = c(2, 3))

tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

Обратите внимание, что тензор заполнялся построчно. В этом состоит отличие от R, который заполняет массивы по столбцам:

```
array(1:6, dim = c(2, 3))

      [,1] [,2] [,3]
[1,]   1   3   5
[2,]   2   4   6
```

Эта разница между упорядочением по строкам и по столбцам (также известная как порядок C и Fortran соответственно) является важной особенностью, о которой нельзя забывать при преобразовании между массивами R и тензорами. Массивы R всегда упорядочены в стиле Fortran, а тензоры TensorFlow всегда упорядочены в стиле C, и это различие нужно учитывать всякий раз, когда вы изменяете форму массива.

Когда вы работаете с тензорами, изменение формы будет опираться на порядок в стиле C. Работая с массивами R, вы можете явно указать порядок заполнения как атрибут `array_reshape()`:

```
array_reshape(1:6, c(2, 3), order = "C")

      [,1] [,2] [,3]
[1,]   1   2   3
[2,]   4   5   6

array_reshape(1:6, c(2, 3), order = "F")

      [,1] [,2] [,3]
[1,]   1   3   5
[2,]   2   4   6
```

Наконец, функции `array_reshape()` и `as_tensor()` позволяют оставить размер одной из осей неопределенным, и он будет авто-

матически вычислен с использованием размера массива и размера остальных осей. Просто укажите -1 или NA для оси, которую вы хотите вывести:

```
array_reshape(1:6, c(-1, 3))
```

```
  [,1] [,2] [,3]
[1,]  1  2  3
[2,]  4  5  6
```

```
as_tensor(1:6, shape = c(NA, 3))
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

### 3.7.2 Срезы тензоров

Подмножество тензора похоже, но не идентично подмножеству массива R. *Срезы тензоров* предлагают некоторые преимущества, которых нет у массивов R, и наоборот.

Тензоры позволяют выполнять срезы с отсутствующим значением на одном конце диапазона среза, что неявно означает «остальную часть тензора в этом направлении» (массивы R не обладают таким удобным свойством). В примере из главы 2, где мы хотели вырезать кадр из изображений MNIST, мы могли бы вместо этого указать NA для среза:

```
train_images <- as_tensor(dataset_mnist())$train$x
my_slice <- train_images[, 15:NA, 15:NA]
```

Имейте в виду, что выражение 15:NA вызовет ошибку R в других контекстах; оно допустимо только в скобках операции тензорного среза.

Также можно использовать отрицательные индексы. Обратите внимание, что в отличие от массивов R отрицательные индексы не удаляют элементы; вместо этого они указывают положение индекса относительно конца текущей оси. (Поскольку это отличается от стандартного поведения подмножества R, при первом обнаружении отрицательного индекса среза выдается предупреждение.) Например, чтобы обрезать изображения до фрагментов размером 14×14 пикселей с центром посередине, вы можете применить следующий код:

```
my_slice <- train_images[, 8:-8, 8:-8]
```

Warning:

Negative numbers are interpreted python-style

➔ when subsetting tensorflow tensors.

See ?`[.tensorflow.tensor` for details.

To turn off this warning,

➔ set `options(tensorflow.extract.warn\_negatives\_pythonic = FALSE)`

Вы также можете использовать специальный объект `all_dims()` всякий раз, когда хотите неявно зафиксировать оставшиеся измерения, не указывая точное количество запятых (,). Например, если вы хотите получить только первые 100 изображений, то можете написать

```
my_slice <- train_images[1:100, all_dims()]
```

вместо

```
my_slice <- train_images[1:100, , ]
```

Это удобно при написании кода, который может работать с тензорами разных рангов, например для сопоставления срезов входных и целевых данных модели в пакетном измерении.

### 3.7.3 Операции с тензорами разной размерности

Операции с тензорами разной размерности (*tensor broadcasting*, *трансляция тензора*) были представлены в главе 2. Трансляция выполняется, когда у нас есть операция над двумя тензорами разного размера и мы хотим, чтобы меньший тензор был приведен (транслирован) к форме большего тензора. Трансляция состоит из следующих двух этапов:

- 1 оси (называемые осями трансляции) добавляются к меньшему тензору до совпадения с `dim` большего тензора;
- 2 меньший тензор повторяют вместе с этими новыми осями до совпадения с полной формой большего тензора.

С трансляцией обычно выполняют поэлементные операции, которые принимают два входных тензора, где один тензор имеет форму  $(a, b, \dots, n, n + 1, \dots, m)$ , а другой имеет форму  $(n, n + 1, \dots, m)$ . Для осей от  $a$  до  $n - 1$  автоматически выполняется трансляция. В следующем примере поэлементная операция  $+$  применяется к двум тензорам разных форм с предварительной автоматической трансляцией:

```
x <- as_tensor(1, shape = c(64, 3, 32, 10))
y <- as_tensor(2, shape = c(32, 10))
z <- x + y ←————— Выход z имеет форму (64, 3, 32, 10), как x
```

Всякий раз, когда нужно указать семантику трансляции в явном виде, вы можете использовать `tf$newaxis` для вставки измерения размера 1 в тензор:

```
z <- x + y[tf$newaxis, tf$newaxis, , ]
```

### 3.7.4 Модуль tf

Тензоры нужно создавать с некоторым начальным значением. Вы можете использовать для создания тензоров функцию `as_tensor()`, но модуль `tf` также содержит множество других функций для соз-

дания тензоров. Например, вы можете создавать тензоры со всеми единицами или со всеми нулями или тензоры значений, взятых из случайного распределения.

```
library(tensorflow)
tf$ones(shape(1, 3))
| tf.Tensor([[1. 1. 1.]], shape=(1, 3), dtype=float32)
tf$zeros(shape(1, 3))
| tf.Tensor([[0. 0. 0.]], shape=(1, 3), dtype=float32)
tf$random$normal(shape(1, 3), mean = 0, stddev = 1)
| tf.Tensor([[ 0.79165614
  -0.35886717  0.13686056]], shape=(1, 3), dtype=float32)
tf$random$uniform(shape(1, 3))
| tf.Tensor([[0.93715847  0.67879045  0.60081327]], shape=(1, 3),
  -dtype=float32)
```

Тензор случайных значений, взятых из равномерного распределения между 0 и 1. Эквивалентно `array(runif(3 * 1, min = 0, max = 1), dim = c(1, 3))`

Обратите внимание, что модуль `tf` предоставляет полный API Python TensorFlow. Следует иметь в виду, что Python API часто ожидает целые числа, тогда как чистый числовой литерал R, такой как `2`, выдает `double` вместо `int`. В R мы можем указать целочисленный литерал, добавив букву `L`, как в `2L`.

```
tf$ones(c(2, 1))
```

← Тип R `double` вызывает здесь ошибку

```
Error in py_call_impl(callable, dots$args, dots$keywords):
  TypeError: Cannot convert [2.0, 1.0] to EagerTensor of dtype int32
```

```
tf$ones(c(2L, 1L))
| tf.Tensor(
  [[1.]
  [1.]], shape=(2, 1), dtype=float32)
```

← Указание целочисленного литерала позволяет избежать ошибки

Работая с модулем `tf`, мы часто будем использовать целочисленные литералы с суффиксом `L` там, где этого требует API Python.

Еще одна вещь, о которой следует помнить, – функции в модуле `tf` используют соглашение о подсчете индексов начиная с 0, то есть первым элементом списка является элемент с индексом 0. Например, если вы хотите взять среднее значение по первой оси двумерного массива, вы должны сделать это следующим образом:

```
m <- as_tensor(1:12, shape = c(3, 4))
tf$reduce_mean(m, axis = 0L, keepdims = TRUE)
| tf.Tensor([[5 6 7 8]], shape=(1, 4), dtype=int32)
```

Однако соответствующие функции R отсчитывают элементы, начиная с 1:

```
mean(m, axis = 1, keepdims = TRUE)
```

```
tf.Tensor([[5 6 7 8]], shape=(1, 4), dtype=int32)
```

Вы можете получить доступ к справке по функциям модуля tf прямо из среды разработки RStudio IDE. Поместите курсор на функцию tf и нажмите **F1**, чтобы открыть веб-страницу с нужным описанием на [www.tensorflow.org](http://www.tensorflow.org).

### 3.7.5 *Неизменность тензоров и переменные*

Существенная разница между массивами R и тензорами TensorFlow заключается в том, что тензоры TensorFlow не поддаются изменению: они постоянны. Например, в R вы можете сделать следующее (листинг 3.1).

#### Листинг 3.1 Массивам R можно присваивать значения

```
x <- array(1, dim = c(2, 2))
x[1, 1] <- 0
```

Попробуйте сделать то же самое в TensorFlow, и вы получите ошибку «EagerTensor object does not support item assignment» (Объект EagerTensor не поддерживает назначение элементов).

#### Листинг 3.2 Тензорам TensorFlow нельзя присваивать значения

```
x <- as_tensor(1, shape = c(2, 2))
x[1, 1] <- 0 ← Это не сработает, потому что тензор нельзя модифицировать
```

```
Error in `[<-tensorflow.tensor`(`*tmp*`, 1, 1, value = 0):
  TypeError: 'tensorflow.python.framework.ops.EagerTensor'
  object does not support item assignment
```

Чтобы обучить модель, нам нужно обновить ее состояние, которое представляет собой набор тензоров. Если тензоры не поддаются изменению, как нам это сделать? Вот тут-то и появляются *переменные*. `tf$Variable` – это класс, предназначенный для управления изменяемым состоянием в TensorFlow. Вы уже кратко видели его в действии в реализации цикла обучения в конце главы 2.

Чтобы создать переменную, нужно задать некоторое начальное значение, например случайный тензор.

#### Листинг 3.3 Создание переменной TensorFlow

```
v <- tf$Variable(initial_value = tf$random$normal(shape(3, 1)))
v
```

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[ -1.1629326 ],
       [  0.53641343],
       [  1.4736737 ]], dtype=float32)>
```

Состояние переменной можно изменить по необходимости с помощью ее метода `assign` следующим образом.

#### Листинг 3.4 Присвоение значения переменной TensorFlow

```
v$assign(tf$ones(shape(3, 1)))
```

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

Этот метод также работает для подмножества коэффициентов.

#### Листинг 3.5 Присвоение значения подмножеству переменной TensorFlow

```
v[1, 1]$assign(3)
```

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[3.],
       [1.],
       [1.]], dtype=float32)>
```

Точно так же `assign_add()` и `assign_sub()` являются функциональными эквивалентами  $x \leftarrow x + \text{value}$  и  $x \leftarrow x - \text{value}$ .

#### Листинг 3.6 Использование `assign_add()`

```
v$assign_add(tf$ones(shape(3, 1)))
```

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[4.],
       [2.],
       [2.]], dtype=float32)>
```

## 3.7.6 Математические операции в TensorFlow

TensorFlow предлагает большой набор тензорных операций для реализации математических формул. Вот несколько примеров.

#### Листинг 3.7 Несколько основных математических операций

```
a <- tf$ones(c(2L, 2L))
b <- tf$square(a) ← Возведение в квадрат
c <- tf$sqrt(a) ← Квадратный корень
d <- b + c ← Сложение двух тензоров (поэлементно)
```

```
e <- tf$matmul(a, b) ← Произведение двух тензоров (обсуждалось в главе 2)
e <- e * d           ← Поэлементное перемножение тензоров
```

Обратите внимание, что некоторые из этих операций вызываются при помощи соответствующих универсальных шаблонов R. Например, `sqrt(x)` фактически вызовет `tf$sqrt(x)`, если `x` является тензором.

Важно отметить, что каждая из упомянутых операций выполняется «на лету»: в любой момент вы можете распечатать текущий результат, как в обычном коде R. Мы называем это *безотлагательным выполнением*.

### 3.7.7 Взгляд на API `GradientTape` с другой стороны

До этого момента TensorFlow был очень похож на базовый R, только с другими именами функций и дополнительными возможностями работы с тензорами. Но есть одна вещь, которую R затрудняется сделать: получить градиент любого дифференцируемого выражения относительно любой точки входных данных. В TensorFlow для этого достаточно открыть область видимости `tf$GradientTape()` с помощью `with()`, применить нужные вычисления к одному или нескольким входным тензорам, и вы получите градиент результата относительно входных данных.

#### Листинг 3.8 Использование `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 способ чаще всего используют для получения градиентов потерь модели по отношению к ее весам: градиенты `gradients <- tape$gradient(loss, weights)`. Вы видели его в действии в главе 2.

До сих пор входные тензоры в `tape$gradient()` были переменными TensorFlow. На самом деле входные данные могут быть любым произвольным тензором. Однако по умолчанию отслеживаются только *обучаемые переменные*. Неизменяемый тензор вам придется вручную пометить как отслеживаемый, вызвав для него функцию `tape$watch()`.

#### Листинг 3.9 Использование `GradientTape` с неизменяемыми тензорными входными данными

```
input_const <- as_tensor(3)
with(tf$GradientTape() %as% tape, {
  tape$watch(input_const)
  result <- tf$square(input_const)
```

```

})
gradient <- tape$gradient(result, input_const)

```

Почему это необходимо? Потому что было бы слишком затратно заранее хранить информацию, необходимую для вычисления градиента чего-либо по отношению к чему-либо. Чтобы не тратить ресурсы впустую, «ленте» нужно знать, что отслеживать. Обучаемые переменные отслеживаются по умолчанию, потому что вычисление градиента потерь по отношению к списку обучаемых переменных является наиболее распространенным использованием градиентной ленты.

Градиентная лента – мощный инструмент, способный даже вычислять градиенты второго порядка, то есть градиент градиента. Например, градиент положения объекта во времени – это скорость этого объекта, а градиент второго порядка – его ускорение.

Если вы отследите положение падающего яблока вдоль вертикальной оси с течением времени и обнаружите, что оно подтверждает уравнение  $\text{position}(\text{time}) = 4.9 * \text{time}^2$ , каково его ускорение? Давайте используем две вложенные градиентные ленты, чтобы выяснить это.

### Листинг 3.10 Использование вложенных градиентных лент для вычисления градиентов 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)
acceleration

```

Мы используем внешнюю ленту для вычисления градиента второго порядка от внутренней ленты. Естественно, ответ  $4,9 * 2 = 9,8$

```
tf.Tensor(9.8, shape=(), dtype=float32)
```

## 3.7.8 Полный пример: линейный классификатор в чистом TensorFlow

К этому времени вы знаете о тензорах, переменных и тензорных операциях и умеете вычислять градиенты, чего достаточно, чтобы построить любую модель машинного обучения на основе градиентного спуска. И это всего лишь середина третьей главы!

На собеседовании по машинному обучению вас могут попросить реализовать линейный классификатор с нуля в TensorFlow: очень простая задача, которая служит фильтром для отбора кандидатов, имеющих минимальный опыт машинного обучения. Давайте пройдем этот фильтр и воспользуемся вашими новыми знаниями о TensorFlow для реализации такого линейного классификатора.

Во-первых, давайте создадим два синтетических набора данных с хорошей линейной разделимостью: два класса точек на 2D-плоскости. Мы будем генерировать каждый класс точек, извлекая их координаты из случайного распределения с определенной матрицей ковариаций и определенным средним значением. Интуитивно понятно, что матрица ковариаций описывает форму облака точек, а среднее значение описывает его положение на плоскости. Мы повторно используем одну и ту же матрицу для обоих облаков точек, но будем применять два разных средних значения – облака точек будут иметь одинаковую форму, но разные положения.

### Листинг 3.11 Генерация двух классов случайных точек на двумерной плоскости

```
num_samples_per_class <- 1000
Sigma <- rbind(c(1, 0.5),
              c(0.5, 1))
negative_samples <-
  MASS::mvrnorm(n = num_samples_per_class,
                mu = c(0, 3), Sigma = Sigma)
positive_samples <-
  MASS::mvrnorm(n = num_samples_per_class,
                mu = c(3, 0), Sigma = Sigma)
```

Генерируем первый класс: 1000 случайных точек на плоскости. Sigma соответствует овальному облаку точек, ориентированному снизу слева направо вверх

Генерируем второй класс точек с другим средним значением и той же матрицей ковариации

В предыдущем коде и `negative_samples`, и `positive_samples` являются массивами формы  $(1000, 2)$ . Объединим их в единый массив с формой  $(2000, 2)$ .

### Листинг 3.12 Объединение двух классов в массив с формой $(2000, 2)$

```
inputs <- rbind(negative_samples, positive_samples)
```

Сгенерируем соответствующие целевые метки – массив нулей и единиц формы  $(2000, 1)$ , где `targets[i, 1]` равно 0, если `inputs[i]` принадлежит классу 1 (и наоборот).

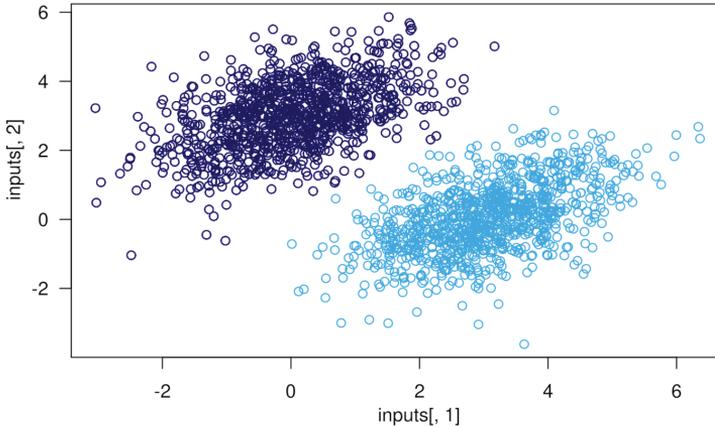
### Листинг 3.13 Генерация соответствующих целей (0 и 1)

```
targets <- rbind(array(0, dim = c(num_samples_per_class, 1)),
                 array(1, dim = c(num_samples_per_class, 1)))
```

Теперь построим визуальное представление наших данных.

### Листинг 3.14 Визуальное представление двух классов точек

```
plot(x = inputs[, 1], y = inputs[, 2],
     col = ifelse(targets[, 1] == 0, "purple", "green"))
```



Теперь давайте создадим линейный классификатор, который научится разделять эти две «кляксы» на плоскости. Линейный классификатор – это аффинное преобразование ( $\text{prediction} = W \cdot \text{input} + b$ ), обученное минимизировать квадрат разницы между прогнозами и целями. Как вы скоро увидите, на самом деле это гораздо более простой пример, чем полный пример демонстрационной двухслойной нейронной сети, который вы видели в конце главы 2. Однако на этот раз вы сможете понять каждую строку кода.

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

### Листинг 3.15 Создание переменных линейного классификатора

```

Точки на плоскости → input_dim <- 2
                        output_dim <- 1 ← Выходные прогнозы – одна оценка для каждой выборки (близкая к 0,
                        W <- tf$Variable(initial_value =                               если ожидается класс 0, и близкая к 1, если ожидается класс 1)
                           tf$random$uniform(shape(input_dim, output_dim)))
                        b <- tf$Variable(initial_value = tf$zeros(shape(output_dim)))

```

Функция прямого прохода показана в листинге 3.16.

### Листинг 3.16 Функция прямого прохода

```

model <- function(inputs)
  tf$matmul(inputs, W) + b

```

Поскольку наш линейный классификатор работает с двумерными входными данными,  $W$  на самом деле представляет собой всего два скалярных коэффициента,  $w_1$  и  $w_2$ :  $W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ . В свою очередь,  $b$  является единственным скалярным коэффициентом. Таким образом, для данной входной точки  $[x, y]$  ее прогнозируемое значение  $\text{prediction} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \cdot [x, y] + b = w_1 * x + w_2 * y + b$ . В следующем листинге показана наша функция потерь.

## Листинг 3.17 Функция потерь (среднеквадратичная ошибка)

```
square_loss <- function(targets, predictions) {
  per_sample_losses <- (targets - predictions)^2
  mean(per_sample_losses)
}
```

`per_sample_losses` будет тензором той же формы, что и `targets` и `predictions`, содержащим оценки потерь по образцам

Нам нужно усреднить эти оценки потерь по образцам в одно скалярное значение потерь; это делает метод `mean()`

Заметим, что в функции `square_loss()` и цели, и прогнозы могут быть тензорами, но не обязательно. Это одна из прелестей интерфейса R – такие обобщения, как `mean()`, `^` и `-`, позволяют вам писать одинаковый код для тензоров и массивов R. Когда `targets` и `predictions` являются тензорами, обобщения переадресуются в функции модуля `tf`. Мы также можем написать эквивалент `square_loss`, используя функции из модуля `tf` напрямую:

```
square_loss <- function(targets, predictions) {
  per_sample_losses <- tf$square(tf$subtract(targets, predictions))
  tf$reduce_mean(per_sample_losses)
}
```

Далее следует этап обучения, на котором модель получает обучающие данные и обновляет веса `W` и `b`, чтобы свести к минимуму функцию потерь на этих данных.

## Листинг 3.18 Функция, реализующая шаг обучения

```
learning_rate <- 0.1
```

```
training_step <- function(inputs, targets) {
  with(tf$GradientTape() %as% tape, {
    predictions <- model(inputs)
    loss <- square_loss(predictions, targets)
  })
  grad_loss_wrt <- tape$gradient(loss, list(W = W, b = b))
  W$assign_sub(grad_loss_wrt$W * learning_rate)
  b$assign_sub(grad_loss_wrt$b * learning_rate)
  loss
}
```

Прямой проход внутри области видимости градиентной ленты

Получение градиента  
потери относительно весов

Обновление весов

Для простоты мы будем проводить *пакетное обучение* вместо *мини-пакетного обучения*: мы будем запускать каждый этап обучения (вычисление градиента и обновление веса) для всех данных, а не перебирать данные небольшими партиями. С одной стороны, это означает, что каждый шаг обучения будет выполняться гораздо дольше, потому что мы будем вычислять прямой проход и градиенты для 2000 образцов одновременно. С другой стороны, каждое обновление градиента будет гораздо эффективнее уменьшать потери на обучающих данных, поскольку оно будет охватывать информа-

цию из всех обучающих образцов, а не из маленького пакета. В результате нам потребуется гораздо меньше циклов обучения, и мы сможем использовать более высокую скорость обучения, чем обычно используем при работе с пакетами (мы будем применять `learning_rate = 0.1`, как определено в листинге 3.18).

### Листинг 3.19 Цикл обучения на полном пакете данных

```
inputs <- as_tensor(inputs, dtype = "float32")
for (step in seq(40)) {
  loss <- training_step(inputs, targets)
  cat(sprintf("Loss at step %s: %.4f\n", step, loss))
}
```

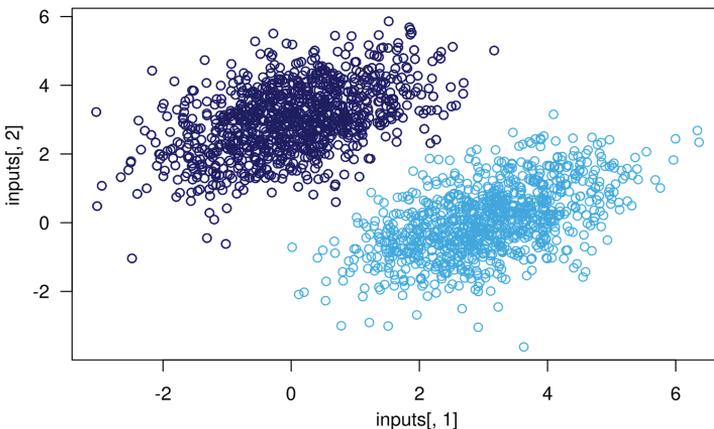
```
Loss at step 1: 0.7263
Loss at step 2: 0.0911
...
Loss at step 39: 0.0271
Loss at step 40: 0.0269
```

После 40 циклов потери при обучении стабилизировались на уровне 0,025. Давайте построим визуализацию того, как наша линейная модель классифицирует точки обучающих данных. Поскольку нашими целями являются нули и единицы, входная точка будет классифицироваться как 0, если ее прогноз ниже 0,5, и как 1, если прогноз выше 0,5:

```
predictions <- model(inputs)

inputs <- as.array(inputs)
predictions <- as.array(predictions)
plot(inputs[, 1], inputs[, 2],
      col = ifelse(predictions[, 1] <= 0.5, "purple", "green"))
```

Преобразование тензоров в массивы R  
для построения визуализации



Напомним, что значение прогноза для данной точки  $[x, y]$  – это просто  $\text{prediction} == [[w1], [w2]] \cdot [x, y] + b == w1 * x + w2 * y + b$ . Таким образом, класс 1 определяется как  $(w1 * x + w2 * y + b) < 0.5$ , а класс 2 определяется как  $(w1 * x + w2 * y + b) > 0.5$ . Несложно заметить, что перед вами уравнение линии на двумерной плоскости:  $w1 * x + w2 * y + b = 0.5$ . Над линией находится класс 1, а под линией – класс 0. Возможно, вы привыкли видеть линейные уравнения в формате  $y = a * x + b$ ; в этом формате наша строка кода выглядит так:  $y = -w1 / w2 * x + (0.5 - b) / w2$ .

Построим эту линию:

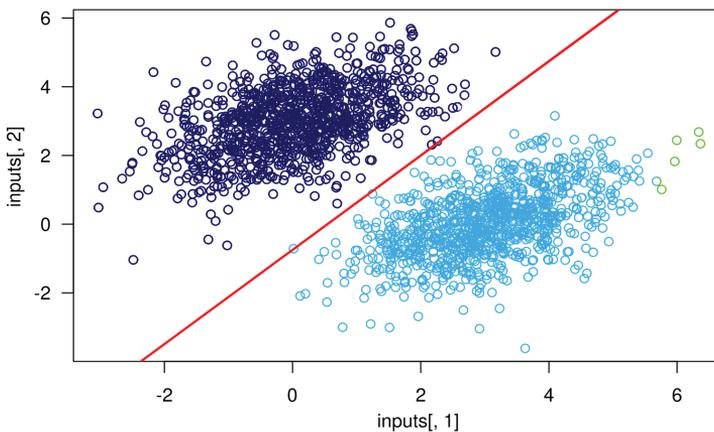
```
plot(x = inputs[, 1], y = inputs[, 2],
     col = ifelse(predictions[, 1] <= 0.5, "purple", "green"))
```

Построение прогнозов  
нашей модели ←

```
slope <- -W[1, ] / W[2, ]
intercept <- (0.5 - b) / W[2, ]
abline(as.array(intercept), as.array(slope), col = "red")
```

← Построение  
линии

| Это значения для уравнения,  
описывающего линию



Именно в этом и состоит суть линейного классификатора: найти параметры линии (или, в многомерных пространствах, гиперплоскости), аккуратно разделяющей два класса данных.

## 3.8 Анатомия нейронной сети и основы API Keras

К этому моменту вы уже знакомы с основами TensorFlow и можете использовать его для реализации несложной модели с нуля – например, пакетного линейного классификатора из предыдущего раздела или нейронной сети из главы 2. Это прочный фундамент, на котором можно смело строить остальное здание. Пришло время перейти

к более продуктивному и надежному инструменту глубокого обучения – API Keras.

### 3.8.1 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: они имеют состояние в виде *весов* слоя, одного или нескольких тензоров, обучаемых с применением алгоритма стохастического градиентного спуска, которые вместе хранят *знание* сети.

Разным слоям соответствуют тензоры разных форматов и разные виды обработки данных. Например, простые векторные данные, хранящиеся в двумерных тензорах с формой (*samples, features*), часто обрабатываются *плотно связанными* слоями, которые также называют *полносвязными*, или *плотными*, слоями (функция `layer_dense` в Keras). Ряды данных хранятся в трехмерных тензорах с формой (*samples, timesteps, features*) и обычно обрабатываются *рекуррентными* слоями, такими как `layer_lstm()`, или одномерными сверточными слоями (`layer_conv_1d()`). Изображения хранятся в четырехмерных тензорах и обычно обрабатываются двумерными сверточными слоями (`layer_conv_2d()`).

Слои можно считать кубиками LEGO глубокого обучения. Фреймворки наподобие Keras делают это сравнение еще более явным. Создание моделей глубокого обучения в Keras осуществляется путем объединения совместимых слоев в конвейеры обработки данных.

#### КЛАСС LAYER В KERAS

Простой API должен иметь единую абстракцию, вокруг которой сосредоточено все остальное. В Keras это класс `Layer`. Иными словами, в Keras все является либо объектом класса `Layer`, либо чем-то, что тесно взаимодействует с этим классом.

`Layer` – это объект, который инкапсулирует некоторое состояние (веса) и некоторые вычисления (прямой проход). Веса обычно назначаются в методе `build()` (хотя они также могут быть созданы в методе `initialize()`), а вычисление определяется в методе `call()`.

В предыдущей главе мы реализовали функцию `layer_naive_dense()`, содержащую два веса,  $W$  и  $b$ , и применили вычисление  $output = activation(dot(input, W) + b)$ . Вот как тот же слой будет выглядеть в Keras.

#### Листинг 3.20 Реализация полносвязного слоя в виде класса Keras Layer

```
layer_simple_dense <- new_layer_class(
  classname = "SimpleDense",
```

```

initialize = function(units, activation = NULL) {
  super$initialize()
  self$units <- as.integer(units)
  self$activation <- activation
},

build = function(input_shape) {
  input_dim <- input_shape[length(input_shape)]
  self$W <- self$add_weight(
    shape = c(input_dim, self$units),
    initializer = "random_normal")
  self$b <- self$add_weight(
    shape = c(self$units),
    initializer = "zeros")
},

call = function(inputs) {
  y <- tf$matmul(inputs, self$W) + self$b
  if (!is.null(self$activation))
    y <- self$activation(y)
  y
}
)

```

Веса создаются в методе build()  
 Получаем последнее измерение  
 add\_weight() – это сокращенный метод для создания весов. Также можно создавать автономные переменные и назначать их как атрибуты слоя, например self\$W <- tf\$Variable(tf\$random\$normal(w\_shape))  
 Определяем вычисления прямого прохода в методе call()

На этот раз вместо создания пустой среды R мы используем функцию `new_layer_class()`, предоставленную Keras. Функция `new_layer_class()` возвращает генератор экземпляров слоя, как и `layer_naive_dense()` в главе 2, но также предоставляет некоторые дополнительные удобные функции (например, компоновку `%>%`, о которой мы расскажем чуть позже).

В следующем разделе мы подробно рассмотрим назначение методов `build()` и `call()`. Не волнуйтесь, если вам пока не все понятно!

Слои можно создать, просто вызвав функцию Keras, которая начинается с префикса `layer_`. Затем, после создания экземпляра слоя, его можно использовать точно так же, как функцию, предоставляя в качестве входных данных тензор TensorFlow:

```

my_dense <- layer_simple_dense(units = 32,
                               activation = tf$nn$relu)
input_tensor <- as_tensor(1, shape = c(2, 784))
output_tensor <- my_dense(input_tensor)
output_tensor$shape
# TensorShape([2, 32])

```

Создание экземпляра слоя, определенного ранее  
 Создание тестовых входных данных  
 Вызов слоя как функции для входных данных

Вам, наверное, интересно, зачем нам понадобилось реализовать `call()` и `build()`, потому что мы закончили тем, что использовали наш слой, просто вызвав его, словно функцию? Дело в том, что нам нужно иметь возможность создавать состояние слоя в нужный момент. Давайте посмотрим, как это работает.

## АВТОМАТИЧЕСКИЙ ВЫВОД ФОРМЫ: СОЗДАНИЕ СЛОЕВ НА ЛЕТУ

Как и в случае с кубиками LEGO, вы можете «соединять» только совместимые слои. Понятие совместимости слоев в данном случае отражает лишь тот факт, что каждый слой принимает и возвращает тензоры определенной формы. Взгляните на следующий пример:

```
layer <- layer_dense(units = 32, activation = "relu")
```

← Полносвязный слой с 32-мерным выходом

Этот слой вернет тензор, в котором первое измерение было преобразовано в 32. Он может быть подключен только к нисходящему слою, который ожидает 32-мерные векторы в качестве входных данных.

Фреймворк Keras избавляет от необходимости беспокоиться о совместимости, потому что уровни, добавляемые в модели, автоматически конструируются так, чтобы соответствовать форме входящего уровня. Например, представьте, что вы написали следующий код:

```
model <- keras_model_sequential(list(
  layer_dense(units = 32, activation = "relu"),
  layer_dense(units = 32)
))
```

Слои не получают никакой информации о форме своих входных данных – вместо этого они автоматически выводят форму входа из формы первых входных данных, которые они увидели. В демонстрационной версии плотного слоя, которую мы реализовали в главе 2 (которую мы назвали `layer_naive_dense()`), нам нужно было явно передать размер слоя в конструктор, чтобы иметь возможность создавать его веса. Это не идеальное решение, потому что оно приводит к моделям, похожим на следующий фрагмент кода, где каждый новый слой должен знать форму предыдущего слоя:

```
model <- model_naive_sequential(list(
  layer_naive_dense(input_size = 784, output_size = 32,
    activation = "relu"),
  layer_naive_dense(input_size = 32, output_size = 64,
    activation = "relu"),
  layer_naive_dense(input_size = 64, output_size = 32,
    activation = "relu"),
  layer_naive_dense(input_size = 32, output_size = 10,
    activation = "softmax")
))
```

Ситуация значительно усложняется, если слой применяет сложные правила для создания выходной формы. Например, как быть, если наш слой возвращает выходные данные формы `if (input_size %% 2 == 0) c(batch, input_size * 2) else c(input_size * 3)`?

Если бы мы переопределили `layer_naive_dense()` как слой Keras, способный автоматически определять форму, он выглядел бы как

предыдущий слой `layer_simple_dense()` (листинг 3.20) с его методами `build()` и `call()`.

В `layer_simple_dense()` мы больше не создаем веса в конструкторе, как в примере `layer_naive_dense()`; вместо этого мы создаем их в специальном методе создания состояния `build()`, который получает в качестве аргумента первую входную форму, видимую слюем. Метод `build()` вызывается автоматически при первом вызове слоя. На самом деле функция, к которой вы обращаетесь при вызове слоя, – это не `call()` напрямую, а некий посредник, который при необходимости сначала вызывает `build()` перед вызовом `call()`.

Функция, которая вызывается при вызове слоя, схематически выглядит так:

```
layer <- function(inputs) {
  if(!self$built) {
    self$build(inputs$shape)
    self$built <- TRUE
  }
  self$call(inputs)
}
```

С автоматическим выводом формы наш предыдущий пример становится простым и аккуратным:

```
model <- keras_model_sequential(list(
  layer_simple_dense(units = 32, activation = "relu"),
  layer_simple_dense(units = 64, activation = "relu"),
  layer_simple_dense(units = 32, activation = "relu"),
  layer_simple_dense(units = 10, activation = "softmax")
))
```

Стоит отметить, что автоматическое определение формы – не единственная работа, которую выполняет класс `Layer`. Он заботится о многих других вещах, в частности о координации между *непосредственным* и *графовым* выполнением (концепция, о которой вы узнаете в главе 7) и маскировании ввода (которое мы рассмотрим в главе 11). А пока просто запомните: при реализации собственных слоев необходимо поместить прямой проход в метод `call()`.

### Компоновка слоев с помощью `%>%` (оператор конвейера)

Хотя вы можете напрямую создавать экземпляры слоев и манипулировать ими, чаще всего все, что вам нужно сделать, – это скомпоновать новый экземпляр слоя с чем-то, например с последовательной моделью. По этой причине первым аргументом всех функций генератора `layer_` является объект. Если указан объект, создается новый экземпляр слоя, который сразу же компоуется с объектом.

Раньше мы создавали модель `keras_model_sequential()`, передавая список слоев, но мы также можем создать модель, добавляя по одному слою за раз:

```
model <- keras_model_sequential()
layer_simple_dense(model, 32, activation = "relu")
layer_simple_dense(model, 64, activation = "relu")
layer_simple_dense(model, 32, activation = "relu")
layer_simple_dense(model, 10, activation = "softmax")
```

В этом случае, поскольку модель предоставляется в качестве первого аргумента для `layer_simple_dense()`, слой создается и сразу компонуется с моделью путем вызова `model$add(layer)`. Обратите внимание, что модель изменяется в процессе выполнения – нам не нужно сохранять вывод наших вызовов `layer_simple_dense()` при компоновке слоев таким способом:

```
length(model$layers)
```

| [1] 4

Одна тонкость заключается в том, что когда конструктор слоя компонуется слой с объектом, он возвращает результат компоновки, а не экземпляр слоя. Таким образом, если в качестве первого аргумента указана модель `keras_model_sequential()`, будет возвращена та же самая модель, только теперь с дополнительным слоем.

Это означает, что вы можете использовать оператор конвейера (`%>%`) для добавления слоев в последовательную модель. Этот оператор входит в пакет `magrittr`. Фактически это сокращенная запись передачи значения слева в качестве первого аргумента функции справа.

Вы можете использовать оператор `%>%` с Keras следующим образом:

```
model <- keras_model_sequential() %>%
  layer_simple_dense(32, activation = "relu") %>%
  layer_simple_dense(64, activation = "relu") %>%
  layer_simple_dense(32, activation = "relu") %>%
  layer_simple_dense(10, activation = "softmax")
```

В чем разница между этим кодом и вызовом `keras_model_sequential()` со списком слоев? Разницы нет – при обоих подходах вы получите одну и ту же модель.

Использование `%>%` приводит к тому, что код становится более читаемым и компактным, поэтому мы будем использовать этот оператор на протяжении всей книги. В RStudio вы можете вставить `%>%` с помощью сочетания клавиш **Ctrl+Shift+M**. Чтобы узнать больше об операторе конвейера, посетите сайт <http://r4ds.had.co.nz/pipes.html>.

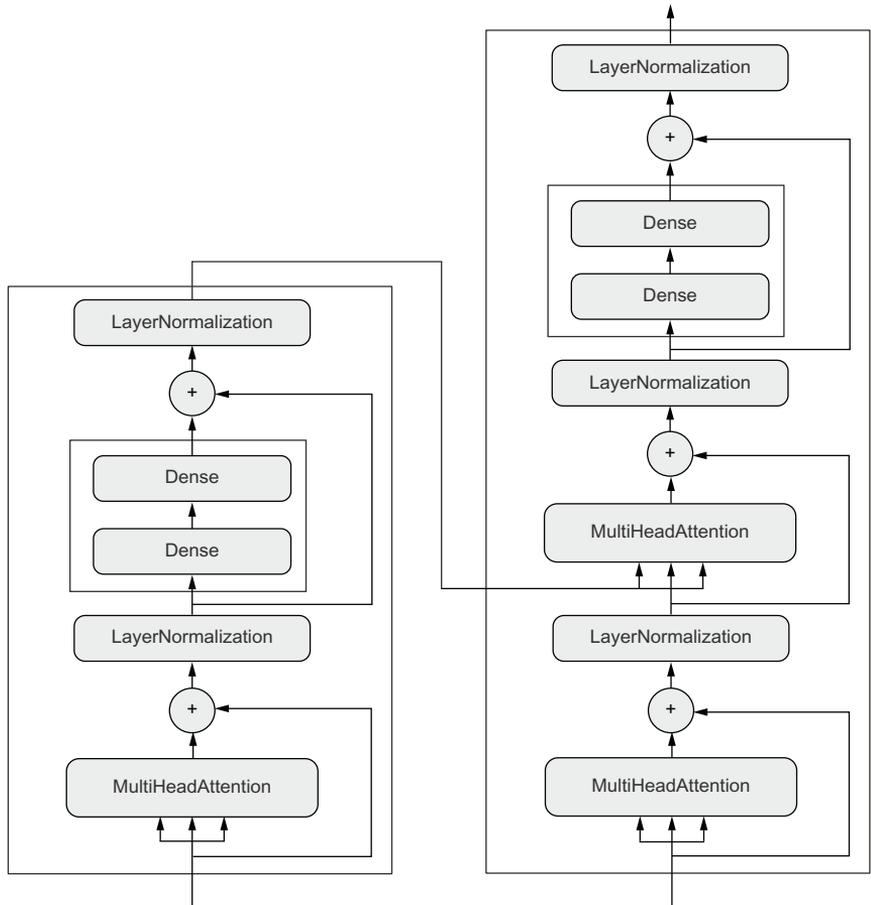
### 3.8.2 От слоев к моделям

Модель глубокого обучения представляет собой граф слоев. В Keras это тип `Model`. До сих пор вы видели только последовательные модели, которые представляют собой простые наборы слоев, отображающие один вход на один выход. Но на практике вы столкнетесь

с гораздо более широким разнообразием сетевых топологий. Вот некоторые из них:

- сети с двумя ветвями (two-branch networks);
- многоголовые сети (multihead networks);
- остаточные соединения (residual connections).

Топология сети может быть очень сложной. Например, на рис. 3.2 показана топология графа слоев Transformer, общей архитектуры, предназначенной для обработки текстовых данных.



**Рис. 3.2** Архитектура Transformer (описана в главе 11). В ней происходят сложные процессы. В следующих нескольких главах вы будете последовательно продвигаться к ее пониманию

В целом в Keras есть два способа создания таких моделей: вы можете напрямую определить `new_model_class()` или использовать функциональный API, который позволяет делать больше с меньшим количеством кода. Мы рассмотрим оба подхода в главе 7.

Топология модели определяет *пространство гипотез*. Возможно, вы помните, что в главе 1 мы описали машинное обучение как поиск полезных представлений некоторых входных данных в заранее определенном пространстве возможностей, используя сигнал обратной связи. Выбирая топологию сети, вы ограничиваете свое пространство возможностей (пространство гипотез) определенной серией тензорных операций, отображая входные данные в выходные данные. Затем вы будете искать хороший набор значений весовых тензоров, задействованных в этих тензорных операциях.

Чтобы учиться на данных, вы должны делать предположения о них. Эти предположения определяют, чему можно научиться. Таким образом, структура вашего пространства гипотез – архитектура вашей модели – чрезвычайно важна. Она кодирует предположения, которые вы делаете о своей проблеме, – предварительные знания, с которых начинается модель. Например, если вы работаете над задачей классификации двух классов с моделью, состоящей из одного `layer_dense()` без активации (чистое аффинное преобразование), вы предполагаете, что ваши два класса линейно разделимы.

Выбор правильной сетевой архитектуры – это больше искусство, чем наука, и хотя вы можете положиться на некоторые передовые методы и принципы, только практический опыт поможет вам стать настоящим архитектором нейронной сети. Следующие несколько глав научат вас принципам построения нейронных сетей и помогут развить интуитивное чутье в отношении того, какая архитектура лучше сработает для конкретной задачи. После прочтения этих глав у вас должно сформироваться четкое понимание того, какие типы архитектур моделей подходят для различных типов задач, как построить эти сети на практике, как выбрать правильную конфигурацию обучения и как настроить модель, чтобы получить от нее желаемые результаты.

### 3.8.3 Этап «компиляции»: настройка процесса обучения

После того как архитектура модели определена, вам нужно выбрать еще три вещи:

- *функция потерь (целевая функция)* – величина, которая будет минимизирована во время обучения. Она представляет собой меру успеха для поставленной задачи;
- *оптимизатор* – определяет, как сеть будет обновляться на основе функции потерь. Он реализует определенный вариант стохастического градиентного спуска (SGD);
- *метрики (показатели)* – меры успеха, которые вы будете отслеживать во время обучения и проверки, например точность классификации. В отличие от потери, обучение не оптимизирует напрямую для этих показателей. Таким образом, показатели не обязательно должны быть дифференцируемыми.

После того как вы выбрали функцию потерь, оптимизатор и метрики, вы можете использовать методы `compile()` и `fit()`, чтобы начать обучение вашей модели. В качестве альтернативы вы также можете написать свои собственные обучающие циклы – мы расскажем, как это сделать, в главе 7. Это намного более трудоемкое занятие! А пока давайте взглянем поближе 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 `optimizer`, `loss` и `metrics`:

Задаем линейный классификатор → <code>model &lt;- keras_model_sequential() %&gt;% layer_dense(1)</code> <code>model %&gt;% compile(optimizer = "rmsprop",</code>	Задаем оптимизатор по имени: RMSprop (не чувствителен к регистру) ←
→ <code>loss = "mean_squared_error",</code> <code>metrics = "accuracy")</code>	Задаем метрику (или несколько): в данном случае только точность ←
Задаем функцию потерь по имени: mean_squared_error	

### Модификация моделей по месту

Мы используем оператор `%>%` для вызова `compile()`. Мы могли бы написать этап компиляции сети:

```
compile(model,
  optimizer = "rmsprop",
  loss = "mean_squared_error",
  metrics = "accuracy")
```

Использование `%>%` для вызова `compile` связано не столько с компактностью, сколько с синтаксическим напоминанием о важной характеристике моделей Keras: в отличие от большинства объектов, с которыми вы работаете в R, модели Keras изменяются *по месту* (in-place). Это связано с тем, что модели Keras представляют собой ориентированные ациклические графы слоев, состояние которых обновляется во время обучения. Вы применяете некое действие непосредственно к объекту сети. Размещение сети слева от оператора `%>%` и отсутствие сохранения результатов в новую переменную говорит о том, что вы выполняете изменение по месту.

В предыдущем вызове `compile()` мы передали оптимизатор, функцию потерь и метрики как строки (например, "rmsprop"). Эти строки на самом деле являются метками, которые преобразуются в объекты R. Например, "rmsprop" превращается в `optimizer_rmsprop()`. Важно отметить, что эти аргументы также можно указать как экземпляры объекта:

```
model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = loss_mean_squared_error(),
```

```

metrics = metric_binary_accuracy()
)

```

Это полезно, если вы хотите передать свои собственные потери или метрики или если хотите дополнительно настроить используемые вами объекты, например передав оптимизатору аргумент `learning_rate`:

```

model %>% compile(
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  loss = my_custom_loss,
  metrics = c(my_custom_metric_1, my_custom_metric_2)
)

```

В главе 7 мы расскажем, как создавать собственные функции потерь и метрики. Но в общем случае вам не придется создавать свои собственные потери, метрики или оптимизаторы с нуля, потому что Keras предлагает широкий спектр встроенных решений, среди которых с большой вероятностью можно найти то, что вам нужно.

Оптимизаторы:

```

ls(pattern = "^optimizer_", "package:keras")

[1] "optimizer_adadelata" "optimizer_adagrad" "optimizer_adam"
[4] "optimizer_adamax"  "optimizer_nadam"  "optimizer_rmsprop"
[7] "optimizer_sgd"

```

Функции потерь:

```

ls(pattern = "^loss_", "package:keras")

[1] "loss_binary_crossentropy"
[2] "loss_categorical_crossentropy"
[3] "loss_categorical_hinge"
[4] "loss_cosine_proximity"
[5] "loss_cosine_similarity"
[6] "loss_hinge"
[7] "loss_huber"
[8] "loss_kl_divergence"
[9] "loss_kullback_leibler_divergence"
[10] "loss_logcosh"
[11] "loss_mean_absolute_error"
[12] "loss_mean_absolute_percentage_error"
[13] "loss_mean_squared_error"
[14] "loss_mean_squared_logarithmic_error"
[15] "loss_poisson"
[16] "loss_sparse_categorical_crossentropy"
[17] "loss_squared_hinge"

```

Метрики:

```

ls(pattern = "^metric_", "package:keras")

[1] "metric_accuracy"

```

```
[2] "metric_auc"  
[3] "metric_binary_accuracy"  
[4] "metric_binary_crossentropy"  
[5] "metric_categorical_accuracy"  
[6] "metric_categorical_crossentropy"  
[7] "metric_categorical_hinge"  
[8] "metric_cosine_proximity"  
[9] "metric_cosine_similarity"  
[10] "metric_false_negatives"  
[11] "metric_false_positives"  
[12] "metric_hinge"  
[13] "metric_kullback_leibler_divergence"  
[14] "metric_logcosh_error"  
[15] "metric_mean"  
[16] "metric_mean_absolute_error"  
[17] "metric_mean_absolute_percentage_error"  
[18] "metric_mean_iou"  
[19] "metric_mean_relative_error"  
[20] "metric_mean_squared_error"  
[21] "metric_mean_squared_logarithmic_error"  
[22] "metric_mean_tensor"  
[23] "metric_mean_wrapper"  
[24] "metric_poisson"  
[25] "metric_precision"  
[26] "metric_precision_at_recall"  
[27] "metric_recall"  
[28] "metric_recall_at_precision"  
[29] "metric_root_mean_squared_error"  
[30] "metric_sensitivity_at_specificity"  
[31] "metric_sparse_categorical_accuracy"  
[32] "metric_sparse_categorical_crossentropy"  
[33] "metric_sparse_top_k_categorical_accuracy"  
[34] "metric_specificity_at_sensitivity"  
[35] "metric_squared_hinge"  
[36] "metric_sum"  
[37] "metric_top_k_categorical_accuracy"  
[38] "metric_true_negatives"  
[39] "metric_true_positives"
```

В этой книге вы увидите примеры применения многих из этих компонентов.

### 3.8.4 Выбор функции потерь

Чрезвычайно важно выбрать правильную функцию потерь, которая соответствует конечной цели вашей работы. Нейронная сеть будет использовать любой доступный короткий путь, чтобы минимизировать потери, поэтому, если цель не полностью коррелирует с критерием успеха для поставленной задачи, ваша сеть в конечном итоге сделает не совсем то, чего вы хотели. Представьте глупый всемогущий ИИ, обученный с помощью SGD на основе неудачно выбранной

целевой функции: «максимальное среднее благополучие всех живущих людей». Чтобы упростить свою работу, этот ИИ может решить убить всех людей с достатком ниже среднего и сосредоточиться на благополучии оставшихся, потому что среднее благополучие не зависит от количества живущих. Вряд ли вам понравится такое решение! Просто помните, что все построенные вами нейронные сети будут безжалостно снижать свою функцию потерь, поэтому выбирайте цель с умом, иначе вам придется столкнуться с непреднамеренными побочными эффектами.

К счастью, когда дело доходит до распространенных задач, таких как классификация, регрессия и прогнозирование последовательности, вы можете следовать простым рекомендациям, чтобы выбрать правильную потерю. Например, вы будете использовать бинарную кросс-энтропию для задачи классификации двух классов, категориальную кросс-энтропию для задачи классификации многих классов и т. д. Только если вы работаете над действительно новыми исследовательскими задачами, вам придется разрабатывать собственные функции потерь. В следующих нескольких главах мы подробно опишем, какие функции потерь следует выбирать для широкого круга общих задач.

### 3.8.5 Использование метода `fit()`

За `compile()` следует метод `fit()`, реализующий непосредственно цикл обучения. Вот его ключевые аргументы:

- *обучающие данные* (входные и целевые). Обычно их передают в виде массивов R, тензоров или объекта набора данных TensorFlow. Вы узнаете больше об API `tfdatasets` в следующих главах;
- *количество эпох* для обучения – сколько раз цикл обучения должен пройти по набору данных;
- *размер пакета* для использования в каждой эпохе мини-пакетного градиентного спуска – количество обучающих примеров, на которых вычисляют градиент для одного шага обновления веса.

**Листинг 3.21** Вызов метода `fit()` с массивами R

```
history <- model %>%
  fit(inputs,
      targets,
      epochs = 5,
      batch_size = 128)
```

Входные образцы в виде массива R

Соответствующие цели обучения в виде массива R

Обучающий цикл пройдет по данным пять раз

Обучающий цикл будет перебирать данные пакетами по 128 образцов

Вызов `fit()` возвращает объект `history`. Этот объект содержит свойство `metrics`, которое представляет собой именованный спи-

сок значений, состоящий из потерь для каждой эпохи и конкретных имен метрик:

```
str(history$metrics)
```

```
List of 2
 $ loss : num [1:5] 14.2 13.6 13.1 12.6 12.1
 $ binary_accuracy: num [1:5] 0.55 0.552 0.554 0.557 0.559
```

### 3.8.6 Отслеживание потерь и показателей на контрольных данных

Цель машинного обучения состоит не в том, чтобы получить модели, которые хорошо работают на обучающих данных, что легко: все, что вам нужно сделать, – это следовать градиенту. Цель состоит в том, чтобы получить модели, которые *вообще* хорошо работают, особенно в точках данных, с которыми модель никогда раньше не сталкивалась. Тот факт, что модель хорошо работает с обучающими данными, не означает, что она будет хорошо работать с данными, которые она никогда не видела! Например, могло так случиться, что ваша модель в ходе обучения просто *запомнила* соответствие между вашими обучающими выборками и их целями, что бесполезно для задачи прогнозирования по данным, которые модель никогда раньше не видела. Мы рассмотрим этот момент более подробно в главе 5.

Чтобы узнать, как модель работает с новыми данными, принято «откладывать в сторону» некоторую часть обучающих данных. Вы не будете обучать модель на этих данных, но будете использовать их для вычисления показателя потерь, который покажет, насколько хорошо ваша модель работает с незнакомыми данными. Для этого предназначен аргумент `validation_data` в методе `fit()`. Как и обучающие данные, контрольные данные можно передавать в виде массивов R или в виде объекта `Dataset TensorFlow`.

#### Листинг 3.22 Использование аргумента `validation_data`

```
model <- keras_model_sequential() %>%
  layer_dense(1)

model %>% compile(optimizer_rmsprop(learning_rate = 0.1),
                 loss = loss_mean_squared_error(),
                 metrics = metric_binary_accuracy())

n_cases <- dim(inputs)[1]
num_validation_samples <- round(0.3 * n_cases)
val_indices <-
  sample.int(n_cases, num_validation_samples)
```

Генерируем случайные целые числа `num_validation_samples` в диапазоне `[1, n_cases]`

Резервируем 30 % входных целевых и обучающих данных для проверки (мы исключим эти образцы из обучения и зарезервируем их для вычисления потерь и контрольных показателей)

```

val_inputs <- inputs[val_indices, ]
val_targets <- targets[val_indices, , drop = FALSE]
training_inputs <- inputs[-val_indices, ]
training_targets <-
  targets[-val_indices, , drop = FALSE]

model %>% fit(
  training_inputs,
  training_targets,
  epochs = 5,
  batch_size = 16,
  validation_data = list(val_inputs, val_targets)
)

```

Обучающие данные, применяемые для обновления весов модели  
 Передаем drop = FALSE, чтобы метод R array не отбрасывал измерение size-1 и вместо этого возвращал массив с формой (num\_validation\_samples, 1)  
 Проверочные данные, используемые только для отслеживания контрольных потерь и показателей

Значение потерь на контрольных данных называется *контрольными потерями* (потери валидации, проверочные потери), чтобы отличить его от потерь обучения. Обратите внимание, что важно строго разделять данные обучения и данные проверки: цель проверки состоит в том, чтобы отслеживать, действительно ли то, что изучает модель, полезно для новых данных. Если какие-либо проверочные данные были замечены моделью во время обучения, ваша потеря проверки и метрики будут ошибочными.

Обратите внимание, что если вы хотите вычислить потери при проверке и метрики после завершения обучения, то можете вызвать метод `evaluate()`:

```

loss_and_metrics <- evaluate(model, val_inputs, val_targets,
                             batch_size = 128)

```

Метод `evaluate()` будет выполнять итерацию пакетами (размером `batch_size`) по переданным данным и возвращать числовой вектор, где первая запись – это контрольная потеря, а следующие записи – контрольные метрики. Если у модели нет метрик, возвращается только потеря проверки (вектор R длины 1).

### 3.8.7 Использование модели после обучения

После обучения модели ее можно использовать для прогнозирования новых данных. Этот процесс называется *логическим выводом*, или просто *выводом* (inference). Очевидным способом использования является простой вызов модели:

```

predictions <- model(new_inputs)

```

Получает массив R или тензор TensorFlow и возвращает тензор TensorFlow

Однако при этом модель попытается обработать все входные данные в `new_inputs` одновременно, что может оказаться невыполнимым, если объем данных достаточно велик (в частности, для этого

может потребоваться больше памяти, чем есть у вашего графического процессора).

Лучший способ сделать вывод – использовать метод `predict()`. Он будет перебирать данные небольшими пакетами и возвращать массив прогнозов `R`. И в отличие от вызова модели, он также может обрабатывать объекты `Dataset TensorFlow`:

```
predictions <- model %>%
  predict(new_inputs, batch_size = 128) ←
```

Получает массив `R` или набор данных TF и возвращает массив `R`

Например, если мы используем `predict()` для проверки линейной модели, которую обучили ранее, то получим скалярные оценки, которые соответствуют предсказанию модели для каждого входного образца:

```
predictions <- model %>%
  predict(val_inputs, batch_size = 128)
head(predictions, 10)
```

```
      [,1]
[1,] -0.11416233
[2,]  0.43776459
[3,] -0.02436411
[4,] -0.19723934
[5,] -0.24584538
[6,] -0.18628466
[7,] -0.06967193
[8,]  0.19761485
[9,] -0.28266442
[10,] 0.43299851
```

На данный момент это все, что вам нужно знать о моделях `Keras`. Вы готовы перейти к решению реальных задач машинного обучения с помощью `Keras` в следующей главе.

## Краткие итоги главы

- `TensorFlow` – это мощная платформа для числовых вычислений, которая может работать на `CPU`, `GPU` или `TPU`. Фреймворк `TensorFlow` может автоматически вычислять градиент любого дифференцируемого выражения, его можно распространять на множество устройств, и он может экспортировать программы в различные внешние среды выполнения, даже в `JavaScript`.
- `Keras` – это стандартный API для глубокого обучения с помощью `TensorFlow`. Мы будем использовать его на протяжении всей книги.
- Ключевыми объектами `TensorFlow` являются тензоры, переменные, тензорные операции и градиентная лента.

- Базовым типом в Keras является `Layer`. Он инкапсулирует веса и вычисления. Слои `Layer` собираются в *модели*.
- Прежде чем приступить к обучению модели, вам нужно выбрать оптимизатор, функцию потерь и одну или несколько метрик, которые вы указываете с помощью метода `model %>% compile()`.
- Чтобы обучить модель, вы можете использовать метод `fit()`, который выполняет мини-пакетный градиентный спуск. Вы также можете использовать его для отслеживания ваших потерь и метрик на *проверочных данных* – наборе входных данных, которые модель не видела во время обучения.
- После того как ваша модель обучена, используйте метод `model %>% predict()` для создания прогнозов на новых входных данных.

# Примеры работы с нейросетью: классификация и регрессия

---

## ***Эта глава охватывает следующие темы:***

- ваши первые примеры реального процесса машинного обучения;
- решение задачи классификации векторных данных;
- решение задачи непрерывной регрессии с векторными данными.

В этой главе вы начнете использовать нейронные сети на практике. Вы закрепите знания, полученные из глав 2 и 3, и примените их к решению трех новых задач, представляющих наиболее распространенные варианты использования нейронных сетей:

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

Эти вводные примеры станут вашим первым знакомством с реальными рабочими процессами машинного обучения: вы познакомитесь с предварительной обработкой данных, основными принципами выбора архитектуры модели и оценкой модели.

### Краткий словарь классификации и регрессии

В задачах классификации и регрессии применяют много разных специальных терминов. С некоторыми из них вы уже встречались в предыдущих примерах, а остальные мы будем использовать в следующих главах. У этих терминов есть точные определения, относящиеся только к машинному обучению. Постарайтесь их запомнить:

- *образец*, или *ввод*, – одна точка данных, с которой работает ваша модель;
- *прогноз*, или *вывод*, – то, что получается на выходе вашей модели;
- *цель* – истинное значение, которое в идеале должна была предсказать ваша модель на основе внешних данных;
- *ошибка прогноза*, или *потеря*, – мера расстояния между прогнозом вашей модели и целью;
- *классы* – набор возможных меток для выбора в задаче классификации. Например, при классификации изображений кошек и собак «собака» и «кошка» – это два класса;
- *метка* – конкретный экземпляр аннотации класса в задаче классификации. Например, если изображение № 1234 аннотировано (помечено) как относящееся к классу «собака», то «собака» является меткой изображения № 1234;
- *аннотации*, или *эталоны*, – все цели для набора данных, обычно собираемые людьми;
- *бинарная*, или *двоичная классификация*, – задача классификации, в которой каждый входной образец должен быть отнесен к одной из двух взаимоисключающих категорий;
- *многоклассовая классификация* – задача классификации, в которой каждый входной образец должен быть отнесен к одной из более чем двух категорий, например классификация рукописных цифр;
- *классификация с несколькими метками* – задача классификации, в которой каждому входному образцу может быть присвоено несколько меток. Например, какое-то изображение может содержать как кошку, так и собаку и должно быть снабжено как меткой «кошка», так и меткой «собака». Количество меток изображения обычно варьируется;
- *скалярная регрессия* – задача, целью которой является непрерывное скалярное значение. Хорошим примером является прогнозирование цен на жилье: различные возможные цены образуют непрерывное пространство;
- *векторная регрессия* – задача, целью которой является набор непрерывных значений, например непрерывный вектор. Если вы выполняете регрессию по нескольким значениям (например, координатам ограничивающей рамки на изображении), то вы выполняете векторную регрессию;
- *мини-пакет*, или *пакет*, – небольшой набор образцов (обычно от 8 до 128), которые одновременно обрабатываются моделью. Количество образцов часто равно степени двойки, чтобы упростить выделение памяти на графическом процессоре. В процессе обучения мини-пакет используется для вычисления одного обновления градиентного спуска, применяемого к весам 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 как положительные и отрицательные, опираясь на текст отзывов.

### 4.1.1 Набор данных IMDB

Вы будете работать с набором данных IMDB – множеством 50 000 самых разных отзывов о фильмах в интернет-базе кинофильмов (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.

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

```
library(keras)

imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

#### Оператор множественного присваивания (%<-%)

Все наборы данных, что входят в состав Keras, представляют собой вложенные списки обучающих и контрольных данных. В листинге 4.1 мы использовали *оператор множественного присваивания* %<-% из пакета zeallot, чтобы распаковать список в несколько переменных. То же самое можно записать иначе:

```
imdb <- dataset_imdb(num_words = 10000)
train_data <- imdb$train$x
train_labels <- imdb$train$y
```

```
test_data <- imdb$test$x
test_labels <- imdb$test$y
```

Версия с множественным присваиванием предпочтительнее, потому что является более компактной. Оператор %<-% становится доступным автоматически после загрузки пакета R Keras.

Аргумент `num_words=10000` означает, что в обучающих данных будет сохранено только 10 000 слов, наиболее часто встречающихся в обучающем наборе отзывов. Редкие слова будут отброшены. Это позволит вам работать с вектором управляемого размера. Если бы мы не задали ограничение, мы бы работали с 88 585 уникальными словами в обучающих данных, что чересчур много. Многие из этих слов встречаются только в одном образце и поэтому не могут быть осмысленно использованы 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*:

```
str(train_data)
```

```
List of 25000
 $ : int [1:218] 1 14 22 16 43 530 973 1622 1385 65 ...
 $ : int [1:189] 1 194 1153 194 8255 78 228 5 6 1463 ...
 $ : int [1:141] 1 14 47 8 30 31 7 4 249 108 ...
 $ : int [1:550] 1 4 2 2 33 2804 4 2040 432 111 ...
 $ : int [1:147] 1 249 1323 7 61 113 10 10 13 1637 ...
 $ : int [1:43] 1 778 128 74 12 630 163 15 4 1766 ...
 $ : int [1:123] 1 6740 365 1234 5 1156 354 11 14 5327 ...
 $ : int [1:562] 1 4 2 716 4 65 7 4 689 4367 ...
 [list output truncated]
```

```
str(train_labels)
```

```
int [1:25000] 1 0 0 1 0 0 1 0 1 0 ...
```

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

```
max(sapply(train_data, max))
```

```
[1] 9999
```

Для наглядности в листинге 4.2 показано декодирование одного из отзывов в последовательность слов на английском языке:

#### Листинг 4.2 Декодирование отзывов в последовательность слов

```
word_index <- dataset_imdb_word_index()
```

`word_index` – это список именованных элементов, отображающий слова в целочисленные индексы

Получить обратное представление словаря, отображающее индексы в слова

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

```
reverse_word_index <- names(word_index)
names(reverse_word_index) <- as.character(word_index)

decoded_words <- train_data[[1]] %>%
  sapply(function(i) {
    if (i > 3) reverse_word_index[[as.character(i - 3)]]
    else "?"
  })
decoded_review <- paste0(decoded_words, collapse = " ")
cat(decoded_review, "\n")
```

```
? this film was just brilliant casting location scenery story direction
everyone's really suited the part they played and you could just
imagine being there robert ? is an amazing actor and now the same being
director ...
```

### 4.1.2 Подготовка данных

В нейронную сеть нельзя передать списки целых чисел напрямую. Обычно они имеют разную длину, но нейронные сети работают только с пакетами одинакового размера. 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 (`samples`, `max_length`) и затем передать их в первый слой сети, способный обрабатывать такие целочисленные тензоры (слой встраивания `Embedding`, о котором подробнее мы поговорим далее в этой книге);
- выполнить *непосредственное кодирование* (`multi-hot encode`) списков в векторы нулей и единиц. Это может означать, например, преобразование последовательности [3, 5] в 10 000-мерный вектор, все элементы которого равны нулю, кроме элементов с индексами 3 и 5, которые равны единице. Затем их можно передать в `layer_dense()` – полносвязный слой, способный обрабатывать векторизованные данные с вещественными числами.

Мы пойдем по второму пути, с векторизованными данными, которые для ясности создадим вручную (листинг 4.3).

#### Листинг 4.3 Непосредственное кодирование последовательностей целых чисел

```
vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- array(0, dim = c(length(sequences), dimension))
  for (i in seq_along(sequences)) {
    sequence <- sequences[[i]]
    for (j in sequence)
      results[i, j] <- 1
  }
}
```

Создание матрицы нулей с формой (length(sequences), dimension)

Запись 1 в элемент с нужным индексом

```

    }
  results
}
x_train <- vectorize_sequences(train_data)
x_test  <- vectorize_sequences(test_data)

```

Вот как теперь выглядят образцы:

```
str(x_train)
```

```
num [1:25000, 1:10000] 1 1 1 1 1 1 1 1 1 1 ...
```

Нам также нужно векторизовать метки, что делается очень просто преобразованием целочисленных значений в значения с плавающей запятой:

```
y_train <- as.numeric(train_labels)
y_test  <- as.numeric(test_labels)
```

Теперь данные готовы к передаче в нейронную сеть.

### 4.1.3 Создание модели

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

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

- сколько слоев использовать;
- сколько скрытых единиц выбрать для каждого уровня.

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

- два промежуточных слоя с 16 скрытыми единицами в каждом;
- третий слой будет выводить скалярный прогноз в соответствии с *эмоциональной окраской* (`sentiment`) текущего отзыва.

На рис. 4.1 изображена структура модели, а в листинге 4.4 показана реализация Keras, похожая на пример MNIST, который вы видели ранее.

#### Листинг 4.4 Определение модели

```
model <- keras_model_sequential() %>%
  layer_dense(16, activation = "relu") %>%
```

```
layer_dense(16, activation = "relu") %>%
layer_dense(1, activation = "sigmoid")
```

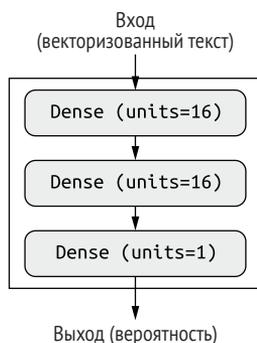


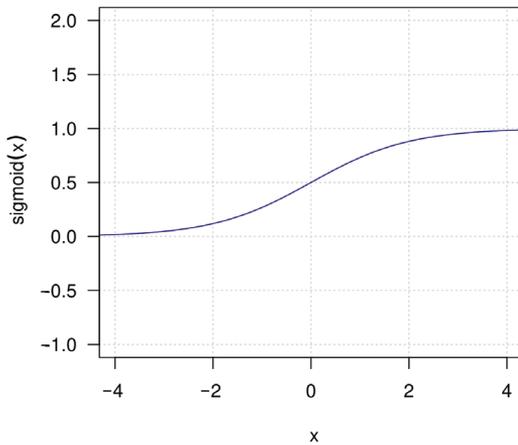
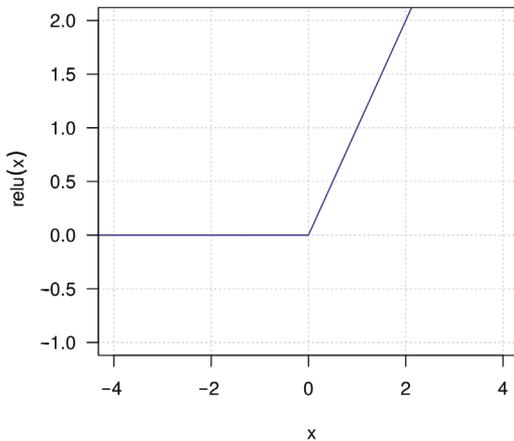
Рис. 4.1 Трехслойная модель

Первый аргумент, передаваемый каждому слою `layer_dense()`, – это количество единиц в слое: размерность пространства представления слоя. В главах 2 и 3 было сказано, что каждый такой `layer_dense()` с активацией `relu` реализует следующую цепочку тензорных операций:

```
output <- relu(dot(input, W) + b)
```

Наличие 16 единиц означает, что матрица весов  $W$  будет иметь форму  $(input\_dimension, 16)$ : скалярное произведение с  $W$  будет проецировать входные данные на 16-мерное пространство представления (а затем вы добавите вектор смещения  $b$  и примените операцию `relu`). Вы можете более наглядно истолковать размерность вашего пространства представления как «количество степеней свободы, которые вы предоставляете модели при изучении внутренних представлений». Наличие большей размерности (многомерное пространство представления) позволяет вашей модели изучать более сложные представления, но делает модель более дорогостоящей в вычислительном отношении и может привести к изучению нежелательных паттернов (шаблонов знаний, которые улучшат точность модели на обучающих данных, но не на тестовых).

Промежуточные уровни используют операцию `relu` в качестве функции активации, а последний слой использует сигмоидную функцию активации и выводит вероятность (оценку вероятности того, что образец относится к классу «1», то есть насколько он близок к положительному отзыву). Функция `relu` (rectified linear unit – блок линейной ректификации, линейно-спрямленная функция) используется для преобразования отрицательных значений в ноль (рис. 4.2), а сигмоидная функция «размазывает» произвольные значения по интервалу  $[0, 1]$  (рис. 4.3), возвращая значения, которые можно интерпретировать как вероятность.

Рис. 4.2 Функция  $\text{relu}$ Рис. 4.3. Функция  $\text{sigmoid}$ 

### Что такое функции активации и зачем они нужны?

Без функции активации, такой как  $\text{relu}$  (также называемой *фактором нелинейности*), полносвязный слой `layer_dense` будет состоять из двух линейных операций – скалярного произведения и сложения:

```
output <- dot(input, W) + 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 уровней представлений друг на друга не приносило бы никакой выгоды, потому что сколь угодно длинная после-

довательность линейных преобразований все равно остается линейным преобразованием – добавление новых уровней не расширяет пространство гипотез (мы говорили об этом в главе 2).

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 вы можете выбирать среди функций активации с немного странными именами: `prelu`, `elu` и т. д.

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

На этом этапе мы выбираем для модели оптимизатор `rmsprop` и функцию потерь `binary_crossentropy`. Обратите внимание, что мы также задали отслеживание точности во время обучения.

#### Листинг 4.5 Компиляция модели

```
model %>% compile(optimizer = "rmsprop",  
                 loss = "binary_crossentropy",  
                 metrics = "accuracy")
```

### 4.1.4 Проверка вашего выбора

Как было сказано в главе 3, модели глубокого обучения ни в коем случае нельзя проверять на их собственных обучающих данных. Для тестирования модели обычно используют проверочный набор. Создадим его, выбрав 10 000 образцов из исходного набора обучающих данных.

#### Листинг 4.6 Создание проверочного набора

```
x_val <- x_train[seq(10000), ]  
partial_x_train <- x_train[-seq(10000), ]  
y_val <- y_train[seq(10000)]  
partial_y_train <- y_train[-seq(10000)]
```

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

#### Листинг 4.7 Обучение модели

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```

На CPU на каждую эпоху будет потрачено менее 2 секунд – обучение закончится через 20 секунд. В конце каждой эпохи обучение приостанавливается, потому что модель вычисляет потери и точность на 10 000 образцов проверочных данных.

Обратите внимание, что вызов `model %>% fit()` возвращает объект `history`, как вы видели в главе 3. Этот объект имеет элемент `metrics`, который представляет собой именованный список, содержащий данные обо всем, что произошло во время обучения. Посмотрим, как он выглядит:

```
str(history$metrics)
```

```
List of 4
 $ loss      : num [1:20] 0.526 0.326 0.241 0.191 0.154 ...
 $ accuracy  : num [1:20] 0.799 0.899 0.921 0.937 0.951 ...
 $ val_loss  : num [1:20] 0.415 0.327 0.286 0.276 0.285 ...
 $ val_accuracy: num [1:20] 0.857 0.876 0.891 0.89 0.886 ...
```

Список `metrics` содержит четыре записи: по одной на метрику, которая отслеживалась во время обучения и во время проверки. Мы воспользуемся методом `plot()` для объекта `history`, чтобы отобразить рядом потери при обучении и проверке, а также точность обучения и проверки (рис. 4.4). Учтите, что ваши собственные результаты могут немного отличаться из-за другой случайной инициализации вашей модели.

```
plot(history)
```

Как видите, на этапе обучения потери снижаются с каждой эпохой, а точность растёт. Именно такое поведение ожидается от оптимизации градиентным спуском – величина, которую вы пытаетесь минимизировать, должна становиться все меньше с каждой итерацией. Но это не относится к потерям и точности на этапе проверки: похоже, что они достигли пика в четвертую эпоху. Это пример того, о чем мы предупреждали выше: модель, показывающая хорошие ре-

зультаты на обучающих данных, не обязательно будет показывать такие же хорошие результаты на данных, которые не видела прежде. Выражаясь точнее, в данном случае наблюдается переобучение: после второй эпохи произошла чрезмерная оптимизация на обучающих данных, и в результате получилось представление, характерное для обучающих данных, не обобщающее данные за пределами обучающего набора.

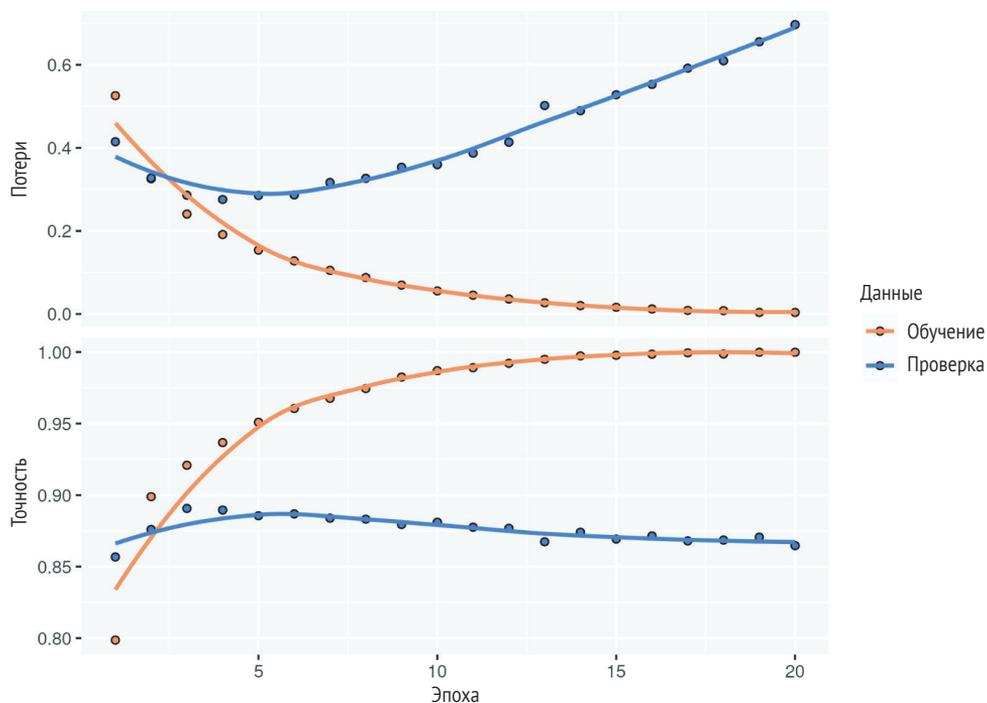


Рис. 4.4 Потери при обучении и проверке и метрики точности

### Отображение истории обучения методом `plot()`

Для создания графиков метод `plot()` объекта с историей обучения использует расширение `ggplot2`, если оно доступно (если нет, для создания графиков используются базовые инструменты). В график включаются все заданные метрики, а также потери; если обучение длилось 10 или более эпох, дополнительно выводятся сглаженные линии. Все эти параметры можно настраивать через различные аргументы метода `plot()`. Если вам понадобится создать собственный нестандартный график, вызовите метод `as.data.frame()` объекта `history`, чтобы получить кадр данных со всеми метриками на этапах обучения и проверки:

```
history_df <- as.data.frame(history)
str(history_df)
```

```
'data.frame': 80 obs. of 4 variables:
 $ epoch : int 1 2 3 4 5 6 7 8 9 10 ...
 $ value : num 0.526 0.326 0.241 0.191 0.154 ...
 $ metric: Factor w/ 2 levels "loss","accuracy": 1 1 1 1 1 1 1 1 1 1 ...
 $ data : Factor w/ 2 levels "training","validation": 1 1 1 1 1 1 1 1 ...
```

В данном случае для предотвращения переобучения можно прервать обучение после третьей эпохи. Вообще говоря, есть целый спектр приемов, ослабляющих эффект переобучения, которые мы рассмотрим в главе 5. А теперь обучим новую модель с нуля в течение четырех эпох и затем оценим ее на контрольных данных (листинг 4.8).

#### Листинг 4.8 Обучение новой модели с нуля

```
model <- keras_model_sequential() %>%
  layer_dense(16, activation = "relu") %>%
  layer_dense(16, activation = "relu") %>%
  layer_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)
```

Окончательный результат выглядит так:

```
results
  loss accuracy
0.2999835 0.8819600
```

Число 0,29 – это потери при тестировании,  
а число 0,88 – это точность при тестировании

Эта простейшая модель позволила достичь точности 88 %. Используя самые современные подходы, можно добиться точности более 95 %.

### 4.1.5 Использование обученной сети для прогнозирования на новых данных

После обучения модели ее можно использовать на практике. Вы можете сгенерировать вероятность того, что отзывы будут положительными, используя метод `predict()`, о котором узнали из главы 3:

```
model %>% predict(x_test)
```

```
      [,1]
[1,] 0.20960191
[2,] 0.99959260
[3,] 0.93098557
```

```
[4,] 0.83782458  
[5,] 0.94010764  
[6,] 0.79225385  
[7,] 0.99964178  
[8,] 0.01294626  
...
```

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

### 4.1.6 Продолжаем эксперименты

Следующие эксперименты помогут вам убедиться, что выбранные параметры архитектуры сети достаточно разумны, хотя есть место для улучшения:

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

### 4.1.7 Промежуточные итоги

Из предыдущего примера можно сделать следующие выводы:

- обычно исходные данные приходится подвергать предварительной обработке, чтобы передать их в нейронную сеть в виде тензоров. Последовательности слов можно преобразовать в бинарные векторы, но существуют также другие варианты;
- стек слоев `layer_dense()` с функцией активации `relu` способен решать широкий круг задач (включая классификацию эмоциональной окраски), и вы, вероятно, чаще всего будете использовать эту комбинацию;
- в задаче бинарной классификации (с двумя выходными классами) ваша модель должна заканчиваться слоем `layer_dense()` одной единицей и функцией активации `sigmoid`: на выходе модели 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 хорошим выбором для любых задач. Этот аспект доставляет меньше всего хлопот;

- улучшая свои показатели на обучающих данных, нейронные сети рано или поздно начинают переобучаться и демонстрируют ухудшение результатов на данных, которые они прежде не видели. Поэтому всегда контролируйте качество работы сети на данных, не входящих в обучающий набор.

## 4.2 Классификация новостных лент: пример многоклассовой классификации

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

В этом разделе 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*; и поскольку каждый экземпляр данных должен быть отнесен только к одному классу, эта задача является примером *однозначной многоклассовой классификации*. Если бы каждый экземпляр данных мог принадлежать нескольким классам (в данном случае тем), эта задача была бы примером *многозначной многоклассовой классификации*.

### 4.2.1 Набор данных Reuters

Мы будем работать с набором данных Reuters, набором новостных лент и их тем, опубликованных агентством Reuters в 1986 году. Это простой набор данных, широко используемых для классификации текста. Существует 46 разных тем; некоторые темы более представительны, некоторые – менее, но для каждой темы в обучающем наборе имеется не менее 10 примеров. Подобно IMDB и MNIST, набор данных Reuters поставляется в составе Keras. Давайте посмотрим, как он выглядит.

#### Листинг 4.9 Загрузка набора данных Reuters

```
reuters <- dataset_reuters(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% reuters
```

По аналогии с примером IMDB, аргумент `num_words = 10000` ограничивает данные 10 000 наиболее часто встречающихся слов. Всего у нас имеется 8982 обучающих и 2246 контрольных образцов:

```
length(train_data)
```

```
[1] 8982
```

```
length(test_data)
```

```
[1] 2246
```

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

```
str(train_data)
```

```
List of 8982
```

```
$ : int [1:87] 1 2 2 8 43 10 447 5 25 207 ...
$ : int [1:56] 1 3267 699 3434 2295 56 2 7511 9 56 ...
$ : int [1:139] 1 53 12 284 15 14 272 26 53 959 ...
$ : int [1:224] 1 4 686 867 558 4 37 38 309 2276 ...
$ : int [1:101] 1 8295 111 8 25 166 40 638 10 436 ...
$ : int [1:116] 1 4 37 38 309 213 349 1632 48 193 ...
$ : int [1:100] 1 56 5539 925 149 8 16 23 931 3875 ...
$ : int [1:100] 1 53 648 26 14 749 26 39 6207 5466 ...
[list output truncated]
```

В листинге 4.10 показано, как можно декодировать новостные записи в слова.

#### Листинг 4.10 Декодирование новостей обратно в текст

```
word_index <- dataset_reuters_word_index()
```

```
reverse_word_index <- names(word_index)
names(reverse_word_index) <- as.character(word_index)
```

```
decoded_words <- train_data[[1]] %>%
  sapply(function(i) {
    if (i > 3) reverse_word_index[[as.character(i - 3)]] ←
    else "?"
  })
```

```
decoded_review <- paste0(decoded_words, collapse = " ")
decoded_review
```

```
[1] "? ? ? said as a result of its december acquisition of space
➡ co it expects ..."
```

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, – это целое число между 0 и 45 – индекс темы:

```
str(train_labels)
```

```
int [1:8982] 3 4 3 4 4 4 4 3 3 16 ...
```

## 4.2.2 Подготовка данных

Для векторизации данных можно повторно использовать код из предыдущего примера.

**Листинг 4.11 Кодирование входных данных**

```
vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for (i in seq_along(sequences))
    results[i, sequences[[i]]] <- 1
  results
}
x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

Векторизовать метки можно двумя способами: сохранить их в тензоре целых чисел или использовать *унитарное кодирование* (one-hot encoding). Этот тип кодирования широко используется для форматирования категорий и также называется *категорийным кодированием* (categorical encoding). В данном случае унитарное кодирование меток заключается в конструировании вектора с нулевыми элементами со значением 1 в элементе, индекс которого соответствует индексу метки (листинг 4.12).

**Листинг 4.12 Кодирование меток**

```
to_one_hot <- function(labels, dimension = 46) {
  results <- matrix(0, nrow = length(labels), ncol = dimension)
  labels <- labels + 1
  for(i in seq_along(labels)) {
    j <- labels[[i]]
    results[i, j] <- 1
  }
  results
}
y_train <- to_one_hot(train_labels)
y_test <- to_one_hot(test_labels)
```

Заметим, что этот способ уже реализован в Keras:

```
y_train <- to_categorical(train_labels)
y_test <- to_categorical(test_labels)
```

## 4.2.3 Построение модели

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

В стеке полносвязных слоев `layer_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 не смогут восстановить ее: каждый слой может стать узким местом для информации. В предыдущем примере мы использовали 16-мерные промежуточные слои, но 16-мерное пространство может оказаться слишком ограниченным для классификации на 46 разных классов: такие малоразмерные слои могут действовать как своеобразные фильтры, постоянно отбрасывая важную информацию. По этой причине в данном примере мы будем использовать уровни с большим количеством измерений. Давайте выберем 64 измерения (листинг 4.13).

### Листинг 3.13 Определение модели

```
model <- keras_model_sequential() %>%  
  layer_dense(64, activation = "relu") %>%  
  layer_dense(64, activation = "relu") %>%  
  layer_dense(46, activation = "softmax")
```

Обратите внимание на две важные особенности этой архитектуры. Во-первых, модель завершается полносвязным слоем `layer_dense()` с размером 46. Это означает, что для каждого входного образца сеть будет выводить 46-мерный вектор. Каждый элемент этого вектора (каждое измерение) представляет один из классов.

Во-вторых, 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 к классу `i`. Сумма 46 элементов всегда будет равна 1.

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

### Листинг 4.14 Компиляция модели

```
model %>% compile(optimizer = "rmsprop",  
  loss = "categorical_crossentropy",  
  metrics = "accuracy")
```

## 4.2.4 Проверка модели

Для тестирования полученной модели создадим проверочный набор, выбрав 1000 образцов из набора обучающих данных.

## Листинг 4.15 Создание проверочного набора

```
val_indices <- 1:1000
x_val <- x_train[val_indices, ]
partial_x_train <- x_train[-val_indices, ]
y_val <- y_train[val_indices, ]
partial_y_train <- y_train[-val_indices, ]
```

Теперь проведем обучение модели в течение 20 эпох.

## Листинг 4.16 Обучение модели

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```

И наконец, выведем графики кривых потерь и точности (рис. 4.5).

## Листинг 4.17 Потери при обучении и проверке и метрики точности

```
plot(history)
```

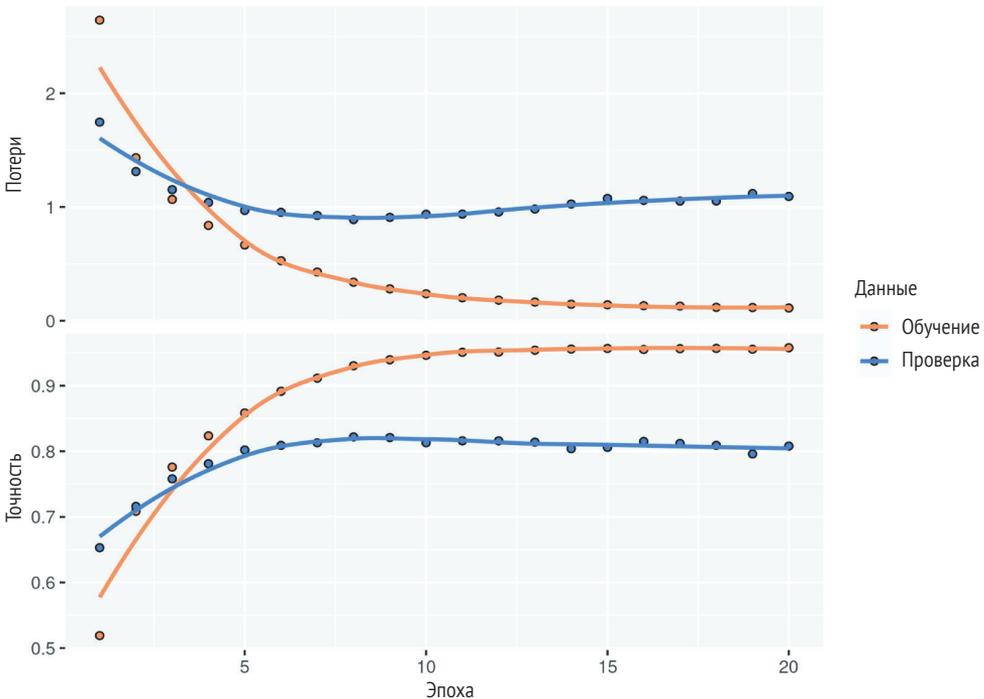


Рис. 4.5 Кривые потерь и точности на проверочном наборе

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

#### Листинг 4.18 Обучение новой модели с нуля

```
model <- keras_model_sequential() %>%
  layer_dense(64, activation = "relu") %>%
  layer_dense(64, activation = "relu") %>%
  layer_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)
```

Так выглядит результат работы модели:

```
results
      loss accuracy
0.9562974 0.7898486
```

Модель достигла точности около 80 %. Со сбалансированной задачей бинарной классификации точность чисто случайного классификатора составила бы 50 %. Но в данном случае у нас 46 классов, и они могут быть представлены неравномерно. Какова будет точность случайного классификатора? Попробуем быстро проверить это эмпирически:

```
mean(test_labels == sample(test_labels))
[1] 0.190561
```

Как видите, точность классификации случайного классификатора составляет около 19 %, и с этой точки зрения результаты нашей модели выглядят весьма неплохо.

## 4.2.5 Предсказания на новых данных

Вызов метода модели `predict()` для новых выборок возвращает распределение вероятностей класса по всем 46 темам для каждой выборки. Сгенерируем прогнозы тем для всех тестовых данных:

```
predictions <- model %>% predict(x_test)
```

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

```
str(predictions)
num [1:2246, 1:46] 0.0000873 0.0013171 0.0094679 0.0001123 0.0001032 ...
```

Сумма элементов этого вектора равна единице:

```
sum(predictions[1, ])
```

```
| [1] 1
```

Элемент с наибольшей вероятностью – это предсказанный класс:

```
which.max(predictions[1, ])
```

```
| [1] 5
```

### 4.2.6 Другой способ обработки меток и потерь

Ранее мы говорили, что метки также можно преобразовать в тензор целых чисел, например так:

```
y_train <- train_labels
y_test <- test_labels
```

Единственное, чем отличается данный подход, – функция потерь. Предыдущая функция потерь `categorical_crossentropy` из листинга 4.18 предполагала, что метки получены методом категорического кодирования. 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 эквивалентна функции `categorical_crossentropy` и просто имеет другой интерфейс.

### 4.2.7 Важность использования достаточно больших промежуточных слоев

Выше уже говорилось, что не следует использовать промежуточные слои менее чем с 46 скрытыми единицами, потому что результат является 46-мерным. Теперь давайте посмотрим, что получится, если ввести узкое место для информации, определив промежуточные слои с размерностями намного меньше 46: например, четырехмерные.

#### Листинг 4.19 Модель с узким местом для информации

```
model <- keras_model_sequential() %>%
  layer_dense(64, activation = "relu") %>%
  layer_dense(4, activation = "relu") %>%
  layer_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 = list(x_val, y_val)
)
```

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

## 4.2.8 Дальнейшие эксперименты

Как и в предыдущем примере, я рекомендую вам провести следующие эксперименты, чтобы натренировать свою интуицию относительно того, какие решения о конфигурации вы должны принимать с такими моделями:

- попробуйте использовать уровни с большим или меньшим числом измерений: 32, 128 и т. д.;
- мы использовали два скрытых слоя. Теперь попробуйте использовать один слой или три.

## 4.2.9 Промежуточные итоги

Из рассмотренного примера мы можем сделать следующие выводы: если мы пытаемся классифицировать образцы данных по  $N$  классам, сеть должна завершаться полносвязным слоем `layer_dense()` с размерностью  $N$ .

В задаче однозначной многоклассовой классификации заключительный слой сети 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 сетью, и истинными распределениями целей.

Метки в многоклассовой классификации можно обрабатывать двумя способами:

- кодировать метки с применением метода кодирования категорий (также известного как непосредственное кодирование) и использовать функцию потерь `categorical_crossentropy`;

- кодировать метки как целые числа и использовать функцию потерь `sparse_categorical_crossentropy`.

Когда требуется классифицировать данные в большое количество категорий, следует избегать создания в сети узких мест для информации в виде слоев с недостаточным 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*, которая заключается в предсказании не дискретной метки, а значения на непрерывной числовой прямой: например, предсказание температуры воздуха на завтра по имеющимся метеорологическим данным или предсказание времени завершения программного проекта по его характеристикам.

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 по таким данным о пригороде в то время, как уровень преступности, ставка местного имущественного налога и т. д. Набор данных, который нам предстоит использовать, имеет интересное отличие от двух предыдущих примеров. Он содержит относительно немного образцов данных: всего 506, разбитых на 404 обучающих и 102 проверочных образца. Каждый признак во входных данных (например, уровень преступности) имеет свой масштаб. Одни признаки имеют значения между 0 и 1, другие – между 1 и 12 и т. д.

#### Листинг 4.20 Загрузка набора данных для Бостона

```
boston <- dataset_boston_housing()
c(c(train_data, train_targets), c(test_data, test_targets)) %<-% boston
```

Посмотрим на данные:

```
str(train_data)
```

```
num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
```

```
str(test_data)
```

```
num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
```

Как видите, у нас имеются 404 обучающих и 102 проверочных образца, каждый с 13 числовыми признаками, такими как уровень преступности, среднее число комнат в доме, удаленность от центральных дорог и т. д. Цели представляют собой медианные значения цен на дома, занимаемые собственниками, в тысячах долларов:

```
str(train_targets)
```

```
num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
```

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

### 4.3.2 Подготовка данных

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

#### Листинг 4.21 Нормализация данных

```
mean <- apply(train_data, 2, mean)
sd <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = sd)
test_data <- scale(test_data, center = mean, scale = sd)
```

Обратите внимание, что величины, используемые для нормализации проверочных данных, вычисляются с использованием обучающих данных. Никогда не используйте в работе какие-либо значения, вычисленные по проверочным данным, даже для таких простых шагов, как нормализация.

### 4.3.3 Построение модели

Из-за небольшого количества образцов мы будем использовать очень маленькую сеть с двумя четырехмерными промежуточными

слоями. Вообще говоря, чем меньше обучающих данных, тем скорее наступит переобучение, а использование маленькой сети – один из способов борьбы с ним.

#### Листинг 4.22 Определение модели

```
build_model <- function() {
  model <- keras_model_sequential() %>%
    layer_dense(64, activation = "relu") %>%
    layer_dense(64, activation = "relu") %>%
    layer_dense(1)

  model %>% compile(optimizer = "rmsprop",
                   loss = "mse",
                   metrics = "mae")

  model
}
```

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

Сеть заканчивается одномерным слоем, не имеющим функции активации (это линейный уровень). 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 уровне применить функцию активации `sigmoid`, сеть обучилась бы предсказывать только значения из диапазона между 0 и 1. В данном случае, с линейным последним слоем, сеть способна предсказывать значения из любого диапазона.

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

Мы также отслеживаем новый параметр 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 этой задаче означает, что в среднем прогнозы отклоняются на 500 долларов США.

### 4.3.4 Оценка качества модели методом K-кратной перекрестной проверки

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. Но так как у нас имеется и без того небольшой набор данных, проверочный набор получился бы слишком маленьким (что-то около 100 образцов). Как следствие оценки при проверке могут сильно меняться в зависимости от того, какие дан-

ные попадут в проверочный и обучающий наборы. Это не позволит надежно оценить качество модели.

Лучшей практикой в таких ситуациях является применение *K-кратной перекрестной проверки* (*K-fold cross-validation*), как показано на рис. 4.6. Суть 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 является оценкой модели. В коде такая проверка реализуется просто.

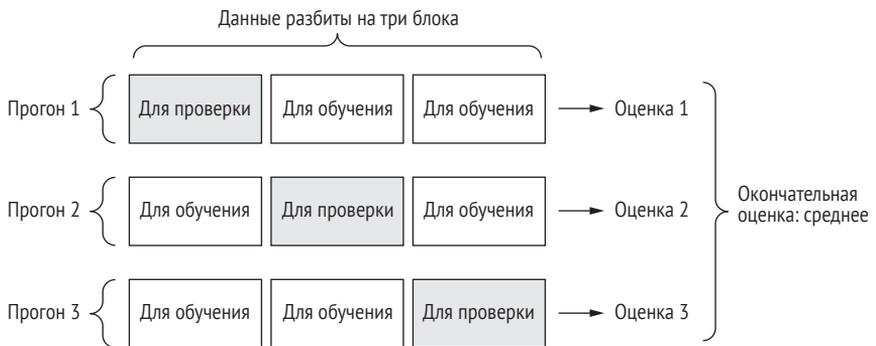


Рис. 4.6 K-кратная перекрестная проверка при K = 3

**Листинг 3.23 K-кратная перекрестная проверка**

```

k <- 4
fold_id <- sample(rep(1:k, length.out = nrow(train_data)))
num_epochs <- 100
all_scores <- numeric()

for (i in 1:k) {
  cat("Processing fold #", i, "\n")
  val_indices <- which(fold_id == i)

  val_data <- train_data[val_indices, ]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices, ]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()

  model %>% fit(
    partial_train_data,
    partial_train_targets,
    epochs = num_epochs,
    batch_size = 16,
    verbose = 0
  )
}

```

Подготовка проверочных данных: данных из блока с номером *k*

Подготовка обучающих данных: данных из остальных блоков

Построение модели Keras (уже скомпилированной)

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

```

results <- model %>%
  evaluate(val_data, val_targets, verbose = 0)
all_scores[[i]] <- results[['mae']]
}

```

Оценка модели на проверочных данных

```

Processing fold # 1
Processing fold # 2
Processing fold # 3
Processing fold # 4

```

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

```

all_scores
[1] 2.435980 2.165334 2.252230 2.362636

mean(all_scores)
[1] 2.304045

```

Разные прогоны действительно показывают разные оценки, от 2,1 до 2,4. Среднее значение (2,3) выглядит более достоверно, чем любая из оценок отдельных прогонов, – в этом главная ценность перекрестной проверки. В данном случае средняя ошибка составила 2300 долларов, что довольно много, если вспомнить, что цены колеблются в диапазоне от 10 000 до 50 000 долларов.

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

#### Листинг 4.24 Сохранение результата проверки перед началом эпохи

```

num_epochs <- 500
all_mae_histories <- list()
for (i in 1:k) {
  cat("Processing fold #", i, "\n")

  val_indices <- which(fold_id == i)
  val_data <- train_data[val_indices, ]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices, ]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()
  history <- model %>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),

```

Подготовка проверочных данных: данных из блока с номером  $k$

Подготовка обучающих данных: данных из остальных блоков

Построение модели Keras (уже скомпилированной)

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

```

epochs = num_epochs, batch_size = 16, verbose = 0
)
mae_history <- history$metrics$val_mae
all_mae_histories[[i]] <- mae_history
}

```

```

Processing fold # 1
Processing fold # 2
Processing fold # 3
Processing fold # 4

```

```
all_mae_histories <- do.call(cbind, all_mae_histories)
```

Теперь можно вычислить средние значения метрики mae для всех прогонов.

#### Листинг 4.25 Создание истории последовательных средних оценок K-кратной проверки

```
average_mae_history <- rowMeans(all_mae_histories)
```

И наконец, построим график (рис. 4.7).

#### Листинг 4.26 Построение графика с результатами проверок

```
plot(average_mae_history, xlab = "epoch", type = 'l')
```

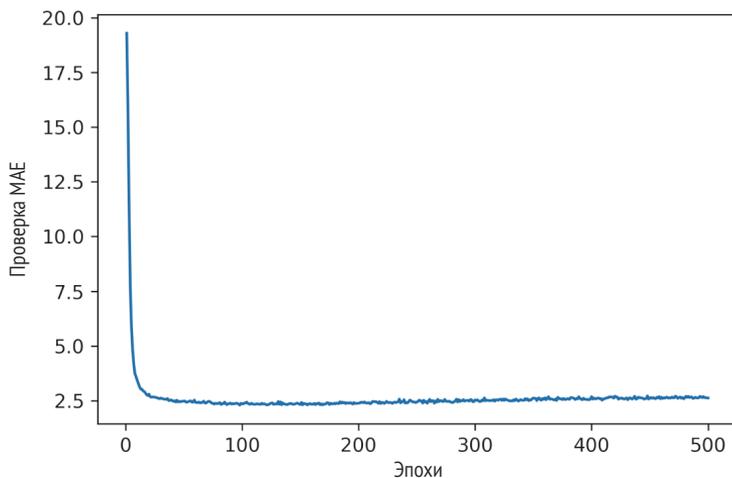


Рис. 4.7 Оценки MAE по эпохам

Из-за проблем с масштабированием и относительно высокой дисперсии может быть немного сложно увидеть общую тенденцию. Значения MAE для первых нескольких эпох значительно больше остальных. Отбросим первые 10 точек данных, которые значительно отличаются от остальной кривой.

#### Листинг 4.27 Построение графика с результатами проверок, за исключением первых 10 точек

```
truncated_mae_history <- average_mae_history[-(1:10)]
plot(average_mae_history, xlab = "epoch", type = 'l',
     ylim = range(truncated_mae_history))
```

Согласно этому графику, оценка MAE значительно улучшается после 100–140 эпох (включая точки, которые мы отбросили). После этого момента начинается переобучение.

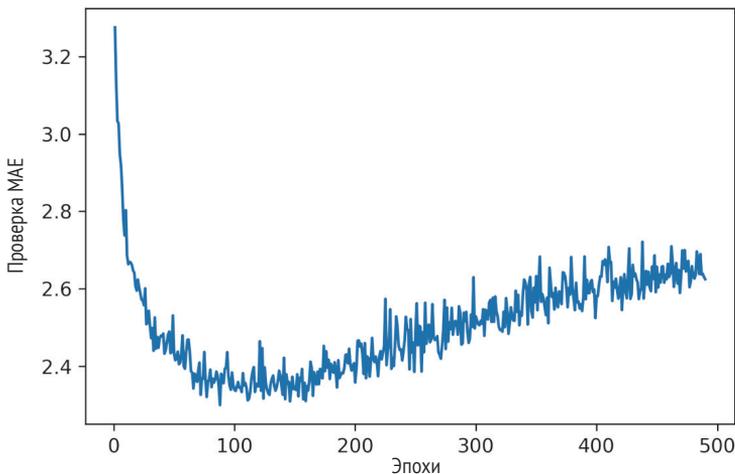


Рис. 4.8 Значения проверки MAE по эпохам, исключая первые 10 точек

Закончив настройку других параметров модели (кроме количества эпох, можно также скорректировать количество промежуточных слоев), можно обучить окончательную версию модели на всех обучающих данных и потом оценить ее качество на контрольных данных.

#### Листинг 4.28 Обучение окончательной версии модели

```
model <- build_model()
model %>% fit(train_data, train_targets,
             epochs = 120, batch_size = 16, verbose = 0)
result <- model %>% evaluate(test_data, test_targets)
```

Получение новой скомпилированной модели

Обучение модели на полных данных

Вот окончательный результат:

```
result["mae"]
```

```
mae
2.476283
```

Средняя ошибка все еще составляет что-то около 2500 долларов. Это уже лучше! Как и в двух предыдущих задачах, вы можете попробовать

изменить количество слоев в модели или количество единиц на слой, чтобы увидеть, сможете ли вы выжать меньшую ошибку теста.

### 4.3.5 Выдача прогнозов на новых данных

При вызове `predict()` в нашей модели бинарной классификации мы получили скалярную оценку от 0 до 1 для каждого входного образца. С помощью нашей модели многоклассовой классификации мы получили распределение вероятностей по всем классам для каждого образца. Теперь, вызывая эту скалярную регрессионную модель, `predict()` возвращает предположение модели о цене образца в тысячах долларов:

```
predictions <- model %>% predict(test_data)
predictions[1, ]
```

```
| [1] 10.27619
```

Модель прогнозирует, что первый дом в проверочном наборе будет стоить около 10 000 долларов.

### 4.3.6 Промежуточные выводы

Из этого примера мы можем сделать следующие выводы:

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

## Краткие итоги главы

- Тремя наиболее распространенными типами задач машинного обучения для векторных данных являются бинарная классификация, многоклассовая классификация и скалярная регрессия.

- Разделы «Промежуточные итоги» ранее в этой главе обобщают важные моменты, которые вы узнали по каждой задаче.
- Регрессия использует другие функции потерь и другие метрики оценки, чем классификация.
- Исходные данные обычно приходится подвергать предварительной обработке перед передачей в нейронную сеть.
- Когда данные включают признаки со значениями из разных диапазонов, их необходимо предварительно масштабировать.
- В процессе обучения модели в некоторый момент наступает эффект переобучения, из-за чего падает качество прогноза на данных, которые она прежде не видела.
- При небольшом объеме входных данных используйте небольшую модель 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  $K$ -кратной перекрестной проверки.

# 5 Основы машинного обучения

---

## ***Эта глава охватывает следующие темы:***

- противоречие между обобщением и оптимизацией – фундаментальная проблема машинного обучения;
- методы оценки моделей машинного обучения;
- рекомендации по улучшению качества обучения модели;
- надежные способы достижения лучшего обобщения.

После трех практических примеров в главе 4 у вас должно сложиться понимание того, как решаются задачи классификации и регрессии с помощью нейронных сетей. Вы собственными глазами увидели главную проблему машинного обучения – переобучение. В этой главе мы подводим прочную теоретическую основу под ваши новые представления о машинном обучении, подчеркивая важность точной оценки модели и баланса между обучением и обобщением.

## ***5.1 Обобщение – цель машинного обучения***

В трех примерах, представленных в главе 4, – прогнозирование оборотов фильмов, классификация новостей и предсказание цен на жилье – мы разделили данные на обучающую, проверочную и конт-

рольную выборки. Причина, по которой нельзя оценивать модели на тех же данных, на которых они обучались, быстро стала очевидной: всего через несколько эпох точность моделей на незнакомых данных начала отличаться в худшую сторону от точности на обучающих данных, которая всегда улучшается по мере обучения. Модели начали переобучаться. Переобучение происходит в каждой задаче машинного обучения.

Фундаментальной проблемой машинного обучения является противоречие между оптимизацией и обобщением. *Оптимизация* представляет собой процесс настройки модели для достижения наилучшего качества прогнозов на обучающих данных (обучение модели), тогда как *обобщение* характеризует то, насколько хорошо обученная модель работает с данными, которые она никогда раньше не видела. Разумеется, смысл обучения модели в том, чтобы получить хорошее обобщение, но... вы не контролируете обобщение; вы можете лишь подогнать модель к обучающим данным. Если вы сделаете это слишком хорошо, возникшее переобучение приведет к резкому снижению обобщающей способности модели.

Но что вызывает переобучение? И как мы можем добиться хорошего обобщения?

### 5.1.1 Недообучение и переобучение

Если вспомнить модели из предыдущей главы, их точность на фиксированных проверочных данных сначала улучшалась по мере обучения, а затем неизбежно достигала пика, после чего начинала ухудшаться. Этот шаблон поведения (показанный на рис. 5.1) является универсальным. Ему следует любая модель с любым набором данных.

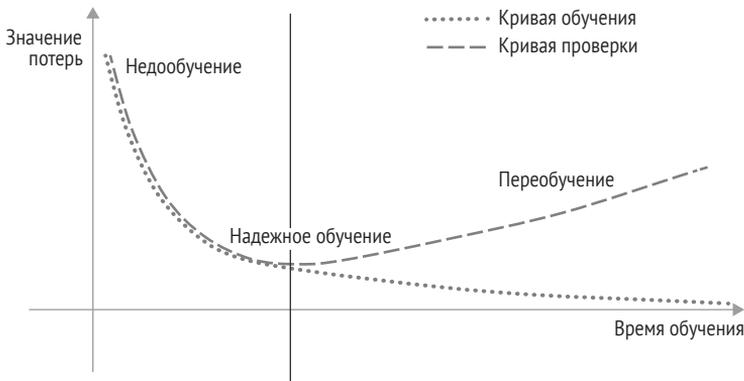


Рис. 5.1 Каноническое явление переобучения

В начале обучения оптимизация и обобщение идут рядом: чем меньше потери на обучающих данных, тем меньше потери на тес-

товых данных. В этот период времени говорят, что ваша модель *недообучена*: она еще может улучшаться, поскольку сеть еще не смоделировала все важные паттерны в обучающих данных. Но после определенного количества итераций на обучающих данных обобщение перестает улучшаться, проверочные метрики перестают расти, а затем постепенно снижаются – начинается переобучение модели. Иными словами, модель начинает изучать глубокие закономерности, характерные исключительно для обучающих данных, но вводящие в заблуждение или нерелевантные, когда речь идет о новых данных.

Риск переобучения особенно велик, когда ваши данные зашумлены, если они связаны с неопределенностью или содержат редкие признаки. Рассмотрим несколько конкретных примеров.

### Зашумленные обучающие данные

В реальных наборах данных довольно часто некоторые входные образцы оказываются недействительными. Например, цифра из набора MNIST может оказаться полностью черным изображением или чем-то вроде артефактов, изображенных на рис. 5.2.

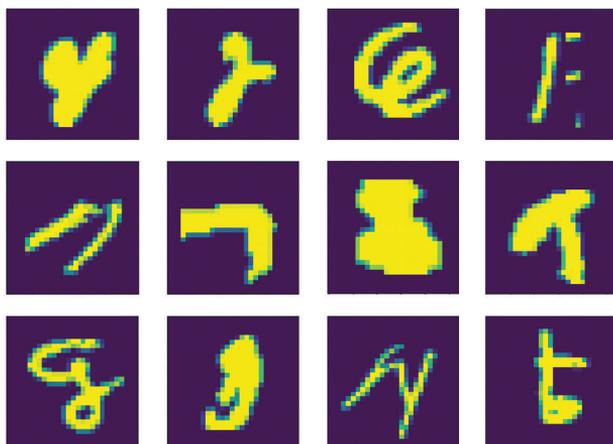


Рис. 5.2 Несколько странных обучающих примеров MNIST

Почему они там оказались? Понятия не имею. Но все они являются частью обучающего набора MNIST. Однако еще хуже то, что совершенно достоверные входные данные в конечном итоге неправильно помечены, как на рис. 5.3.

Если модель будет стараться усвоить такие выбросы, ее способность к обобщению ухудшится, как показано на рис. 5.4. Например, цифра 4, которая очень похожа на ошибочно обозначенную цифру 4 на рис. 5.3, может в конечном итоге быть классифицирована как цифра 9.

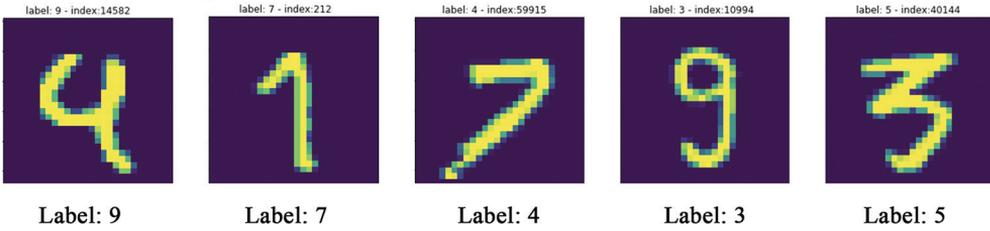


Рис. 5.3 Неправильно помеченные обучающие образцы MNIST

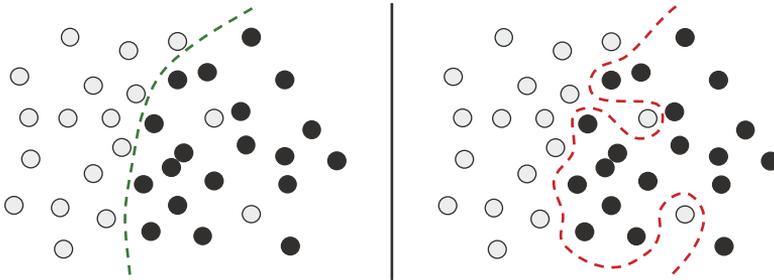


Рис. 5.4 Работа с выбросами: надежное обучение и переобучение

## НЕОДНОЗНАЧНЫЕ ПРИЗНАКИ

Не всегда шум данных возникает из-за неточностей – даже идеально чистые и аккуратно размеченные данные могут быть шумными, когда проблема связана с неопределенностью и двусмысленностью. В задачах классификации часто бывает так, что некоторые области входного пространства признаков одновременно связаны с несколькими классами. Предположим, вы разрабатываете модель, которая получает изображение банана и в ответ предсказывает, является ли банан незрелым, спелым или гнилым. Эти категории не имеют объективных границ, поэтому один и тот же банан может быть по-разному классифицирован разными людьми, размечающими набор данных. Точно так же многие проблемы связаны со случайностью. Допустим, вы используете данные об атмосферном давлении, чтобы предсказать, будет ли завтра дождь, но точно такие же показатели с некоторой вероятностью могут сопровождаться ясным небом.

Модель может перестроиться на такие вероятностные данные, если будет слишком уверена в неоднозначных областях пространства признаков, как на рис. 5.5. Более надежное обучение будет игнорировать отдельные точки данных и рассматривать картину в целом.

## РЕДКИЕ ПРИЗНАКИ И ЛОЖНЫЕ КОРРЕЛЯЦИИ

Если вам довелось видеть за свою жизнь только двух рыжих полосатых кошек и обе они оказались ужасно вредными, вы можете сделать

вывод, что рыжие полосатые кошки обладают дурным характером. Это типичное переобучение: если бы вы изучили побольше кошек, в том числе рыжей масти, вы бы узнали, что окрас кошки плохо коррелирует с характером.

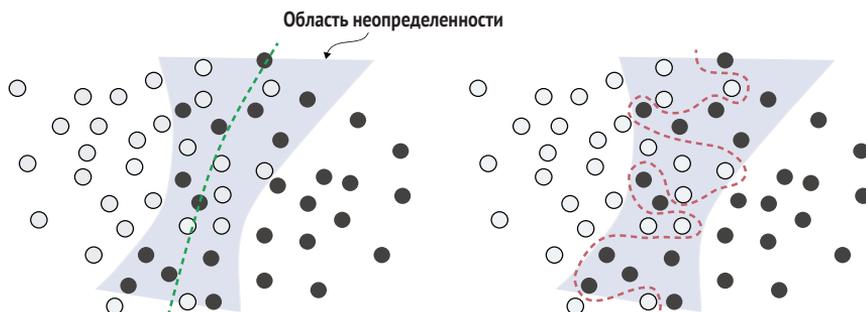


Рис. 5.5 Надежное обучение и переобучение с учетом неоднозначной области пространства признаков

Точно так же модели машинного обучения, обученные на наборах данных, которые содержат редкие значения признаков, очень подвержены переобучению. Например, если в задаче классификации эмоциональной окраски слово *черимойя* (фрукт, произрастающий в Андах) встречается только в одном тексте, и этот текст оказывается отрицательным по тональности, плохо обобщающая модель может придать слишком большой вес этому слову и всегда классифицировать новые тексты, в которых упоминается черимойя, как негативные, хотя объективно в черимойе нет ничего плохого<sup>1</sup>.

Важно отметить, что значение признака не обязательно должно встречаться много раз в одном контексте, чтобы привести к ложным корреляциям. Возьмем слово, которое встречается в 100 образцах ваших обучающих данных и в 54 % случаев связано с положительной эмоциональной окраской, а в 46 % случаев – с отрицательной. Эта небольшая разница вполне могла возникнуть случайно, но ваша модель, скорее всего, научится использовать такой признак в задаче классификации. Это один из самых распространенных источников переобучения.

Вот яркий пример. Используя MNIST, создайте новый обучающий набор, объединив 784 измерения белого шума с существующими 784 измерениями данных. Теперь половина данных представляет собой шум. Для сравнения также создайте эквивалентный набор данных, объединив 784 измерения со всеми нулями. Наша конкатенация бессмысленных признаков не влияет на информативность данных: мы лишь что-то добавляем. Эти преобразования никак не повлияют на точность человеческой классификации.

<sup>1</sup> Марк Твен даже назвал его «самым вкусным фруктом, известным людям».

**Листинг 5.1 Добавление каналов белого шума или нулевых каналов в MNIST**

```
library(keras)

mnist <- dataset_mnist()
train_labels <- mnist$train$y
train_images <- array_reshape(mnist$train$x / 255,
                              c(60000, 28 * 28))

random_array <- function(dim) array(runif(prod(dim)), dim)

noise_channels <- random_array(dim(train_images))
train_images_with_noise_channels <- cbind(train_images, noise_channels)

zeros_channels <- array(0, dim(train_images))
train_images_with_zeros_channels <- cbind(train_images, zeros_channels)
```

Теперь обучим модель из главы 2 на обоих этих обучающих наборах.

**Листинг 5.2 Обучение модели на данных с каналом шума и с нулевым каналом**

```
get_model <- function() {
  model <- keras_model_sequential() %>%
    layer_dense(512, activation = "relu") %>%
    layer_dense(10, activation = "softmax")

  model %>% compile(
    optimizer = "rmsprop",
    loss = "sparse_categorical_crossentropy",
    metrics = "accuracy")

  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)
```

Давайте сравним, как проверочная точность каждой модели меняется с течением времени. Результат показан на рис. 5.6.

## Листинг 5.3 График сравнения проверочной точности

```
plot(NULL,
      main = "Effect of Noise Channels on Validation Accuracy",
      xlab = "Epochs", xlim = c(1, history_noise$params$epochs),
      ylab = "Validation Accuracy", ylim = c(0.9, 1), las = 1)
lines(history_zeros$metrics$val_accuracy, lty = 1, type = "o")
lines(history_noise$metrics$val_accuracy, lty = 2, type = "o")
legend("bottomright", lty = 1:2,
      legend = c("Validation accuracy with zeros channels",
                 "Validation accuracy with noise channels"))
```

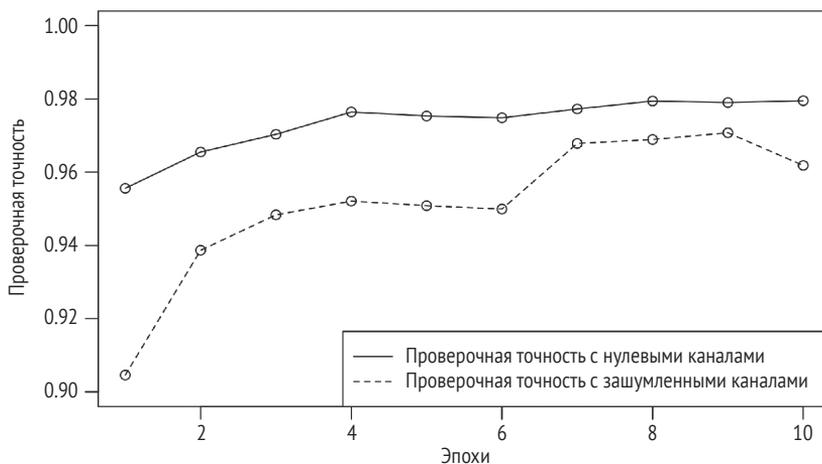


Рис. 5.6 Влияние шумовых каналов на проверочную точность

Несмотря на то что в обоих случаях данные содержат одинаковую информацию, точность модели, обученной на данных с каналом шума, оказывается примерно на один процентный пункт ниже – исключительно из-за влияния ложных корреляций. Чем больше шума вы добавите, тем хуже будет точность.

Зашумленные признаки неизбежно приводят к переобучению. Поэтому в случаях, когда нет полной уверенности, что имеющиеся признаки являются информативными, а не отвлекающими, обычно перед обучением выполняют *отбор признаков*. Например, ограничение данных IMDB до 10 000 самых распространенных слов было грубой формой отбора признаков. Типичный способ отбора признаков заключается в вычислении некоторой *меры полезности* для каждого доступного признака – меры того, насколько информативен признак по отношению к задаче (например, взаимная информация между признаками и метками) – и оставлении только признаков, полезность которых выше некоторого порога. Этот метод позволит отфильтровать каналы белого шума в предыдущем примере.

## 5.1.2 Базовые принципы обобщения в глубоком обучении

Примечательным свойством моделей глубокого обучения является то, что их можно обучить под что угодно, если они имеют достаточную репрезентативную способность.

Не верите? Попробуйте перетасовать метки MNIST и обучить модель на этих странных данных. Несмотря на то что между входными данными и перетасованными метками нет никакой связи, потери при обучении снижаются очень хорошо, даже с относительно небольшой моделью. Естественно, потери при проверке не снижаются со временем, потому что в этом случае нет возможности найти обобщение (см. следующий график).

### Листинг 5.4 Обучение модели MNIST с метками, перемешанными случайным образом

```

                                Используем точку (.) в вызове с несколькими
                                назначениями %<-% для игнорирования некоторых
                                элементов. Здесь мы игнорируем тестовую часть MNIST
c(c(train_images, train_labels), .) %<-% dataset_mnist() ←
train_images <- array_reshape(train_images / 255,
                              c(60000, 28 * 28))

random_train_labels <- sample(train_labels)

model <- keras_model_sequential() %>%
  layer_dense(512, activation = "relu") %>%
  layer_dense(10, activation = "softmax")

model %>% compile(optimizer = "rmsprop",
                 loss = "sparse_categorical_crossentropy",
                 metrics = "accuracy")

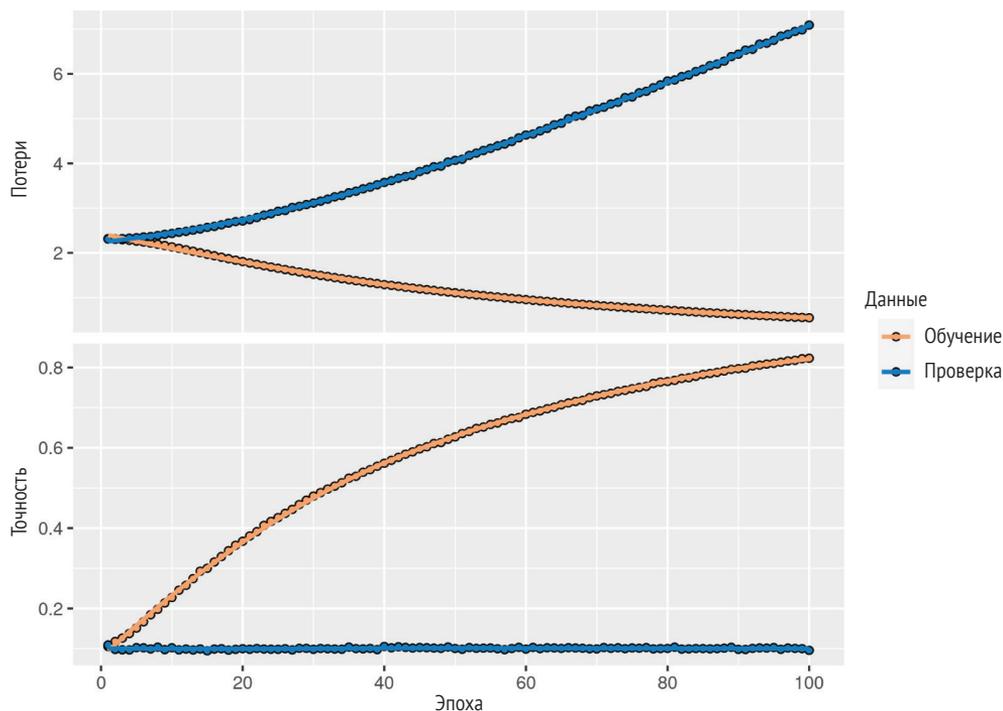
history <- model %>% fit(train_images, random_train_labels,
                       epochs = 100,
                       batch_size = 128,
                       validation_split = 0.2)

```

На самом деле вам даже не нужно возиться с данными MNIST – вы можете просто генерировать входные данные белого шума и случайные метки. Даже на таких данных можно хорошо обучить модель, если у нее достаточно много параметров. Это просто закончилось бы запоминанием определенных входных данных – очень похоже на хеш-таблицу.

Если это так, то почему модели глубокого обучения вообще способны обобщать? Разве они не должны просто изучить специальное сопоставление между обучающими входными данными и целями и составить некую причудливую хеш-таблицу? На каком основании мы ожидаем, что это сопоставление будет работать для новых входных данных?

```
plot(history)
```



Как оказалось, природа обобщения в глубоком обучении имеет мало общего с самими моделями глубокого обучения и намного больше связана со структурой информации в реальном мире. Давайте посмотрим, что здесь происходит на самом деле.

## ГИПОТЕЗА МНОГООБРАЗИЯ

Входные данные классификатора MNIST (до предварительной обработки) представляют собой массив изображений  $28 \times 28$  целых чисел от 0 до 255. Таким образом, общее количество возможных входных значений составляет  $256$  в степени  $784$ , что намного больше, чем количество атомов во Вселенной. Однако очень немногие из этих входных данных будут выглядеть как действительные изображения MNIST: настоящие рукописные цифры занимают лишь крошечное подпространство родительского пространства всех возможных целочисленных массивов  $28 \times 28$ . Более того, это подпространство – не просто набор точек, случайно разбросанных по родительскому пространству; оно имеет очень четкую структуру.

Во-первых, подпространство действительных рукописных цифр *непрерывно*: если взять образец и немного изменить его, он все равно будет распознаваться как та же рукописная цифра. Во-вторых, все выборки в допустимом подпространстве *соединены* гладкими путя-

ми, проходящими через подпространство. Это означает, что если вы возьмете две случайные цифры MNIST A и B, то существует последовательность «промежуточных» изображений, которые превращают A в B, так что две последовательные цифры очень близки друг к другу (рис. 5.7). Возможно, рядом с границей между двумя классами будет несколько неоднозначных фигур, но даже эти фигуры все равно будут очень похожи на цифры.

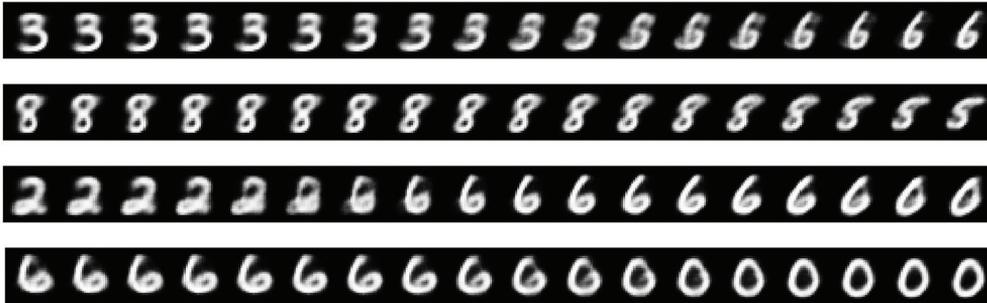


Рис. 5.7 Различные цифры MNIST постепенно переходят друг в друга, показывая, что пространство рукописных цифр образует многообразие. Это изображение было создано с использованием кода из главы 12

С технической точки зрения можно сказать, что рукописные цифры образуют многообразие в пространстве возможных массивов целых чисел  $28 \times 28$ . Звучит как что-то сложное, но идея довольно проста. *Многообразие* – это подпространство меньшей размерности некоторого родительского пространства, локально подобное линейному (евклидову) пространству. Например, гладкая кривая на плоскости представляет собой одномерное многообразие в двумерном пространстве, потому что для каждой точки кривой можно провести касательную (кривая может быть аппроксимирована линией в каждой точке). Гладкая поверхность в трехмерном пространстве представляет собой двумерное многообразие. И т. д.

В более общем смысле *гипотеза многообразия* постулирует, что все естественные данные лежат в маломерном многообразии в многомерном пространстве, где они закодированы. Это довольно сильное утверждение о структуре информации во Вселенной. Насколько нам известно, гипотеза верна, и это причина, по которой глубокое обучение работает. Это справедливо для цифр MNIST, а также для человеческих лиц, морфологии деревьев, звуков человеческого голоса и даже естественного языка.

Гипотеза многообразия подразумевает следующее:

- модели машинного обучения должны обучаться только относительно простым, низкоразмерным, хорошо структурированным подпространствам в их потенциальном входном пространстве (скрытые многообразия);

- внутри одного из этих многообразий всегда можно выполнить *интерполяцию* между двумя входными данными, то есть преобразовать одну точку данных в другую по непрерывному пути, на котором все промежуточные точки попадают на многообразие.

Возможность интерполяции между образцами данных является ключом к пониманию обобщения в глубоком обучении.

## ИНТЕРПОЛЯЦИЯ КАК ИСТОЧНИК ОБОБЩЕНИЯ

Если вы располагаете точками данных, которые можно интерполировать, вы можете начать понимать точки, которые вы никогда раньше не видели, связав их с другими точками, расположенными поблизости на многообразии. Другими словами, вы можете понять всю совокупность пространства, используя только образец пространства. Вы можете использовать интерполяцию, чтобы заполнить пробелы.

Учтите, что интерполяция на скрытом многообразии отличается от линейной интерполяции в родительском пространстве, как показано на рис. 5.8. Например, промежуточное сочетание пикселей между двумя цифрами MNIST обычно не является допустимой цифрой.



Рис. 5.8 Разница между линейной интерполяцией и интерполяцией на скрытом многообразии. Каждая точка данных на скрытом многообразии цифр является действительной цифрой, но промежуточные точки между цифрами обычно таковыми не являются

Хотя глубокое обучение достигает обобщения посредством интерполяции изученной аппроксимации множества данных, было бы ошибкой полагать, что интерполяция – это все, что нужно для обобщения. Это верхушка айсберга. Механизм интерполяции помогает нам шагнуть лишь немного дальше знакомых понятий: он позволяет делать локальные обобщения. Но вот удивительный факт: люди постоянно имеют дело с чрезвычайной новизной, и у нас это прекрасно получается. Вам не нужно заранее обучаться на бесчисленных примерах каждой ситуации, с которой вам когда-либо придется столкнуться. Каждый из ваших дней отличается от любого дня, который вы пережили раньше, и отличается от любого дня, пережитого

кем-либо с момента зарождения человечества. Вы можете спокойно прожить неделю в Нью-Йорке, неделю в Шанхае, а потом неделю в Бангалоре, не нуждаясь в тысячах обучающих примеров жизни и репетициях для каждого города.

Люди способны к далеким обобщениям, которые обеспечиваются когнитивными механизмами, отличными от интерполяции: абстракцией, символическими моделями мира, рассуждениями, логикой, здравым смыслом, врожденными представлениями о мире – тем, что мы обычно называем разумом, в отличие от интуиции и шаблонов распознавания. Последние в значительной степени носят интерполяционный характер, а первые – нет. Оба механизма важны для интеллекта. Подробнее об этом мы поговорим в главе 14.

### ПОЧЕМУ ГЛУБОКОЕ ОБУЧЕНИЕ РАБОТАЕТ

Помните метафору скомканного бумажного листа из главы 2? Лист бумаги представляет собой двумерное многообразие в трехмерном пространстве (рис. 5.9). Модель глубокого обучения – это инструмент для разглаживания бумажных листов, то есть для распутывания скрытых многообразий.

Модель глубокого обучения – это, по сути, очень многомерная кривая, которая является гладкой и непрерывной (с дополнительными ограничениями на ее структуру, исходящими из априорно заданной архитектуры модели), потому что она должна быть дифференцируемой. И эта кривая подгоняется к точкам данных с помощью градиентного спуска, плавно и поэтапно. По самой своей природе глубокое обучение заключается в том, чтобы взять большую сложную кривую – многообразие – и постепенно настраивать ее параметры, пока она не будет соответствовать некоторым точкам обучающих данных.

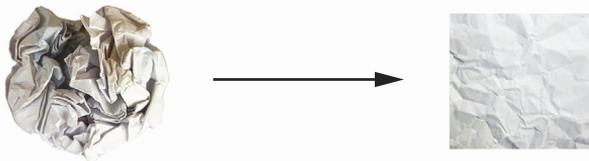


Рис. 5.9 Разборка сложного массива данных

Если кривая включает в себя достаточно много параметров, она может соответствовать чему угодно – действительно, если вы позволите вашей модели обучаться достаточно долго, она фактически просто запомнит свои обучающие данные и вообще не будет обобщать. Однако данные, к которым вы подгоняете модель, не состоят из отдельных точек, разбросанных по базовому пространству. Ваши данные образуют высокоструктурированное низкоразмерное многообразие во входном пространстве – это гипотеза многообразия.

И поскольку подгонка кривой вашей модели к этим данным происходит постепенно и плавно по мере продвижения градиентного спуска, во время обучения модель будет проходить через промежуточное состояние, в котором аппроксимирует естественное многообразие данных, как показано на рис. 5.10.

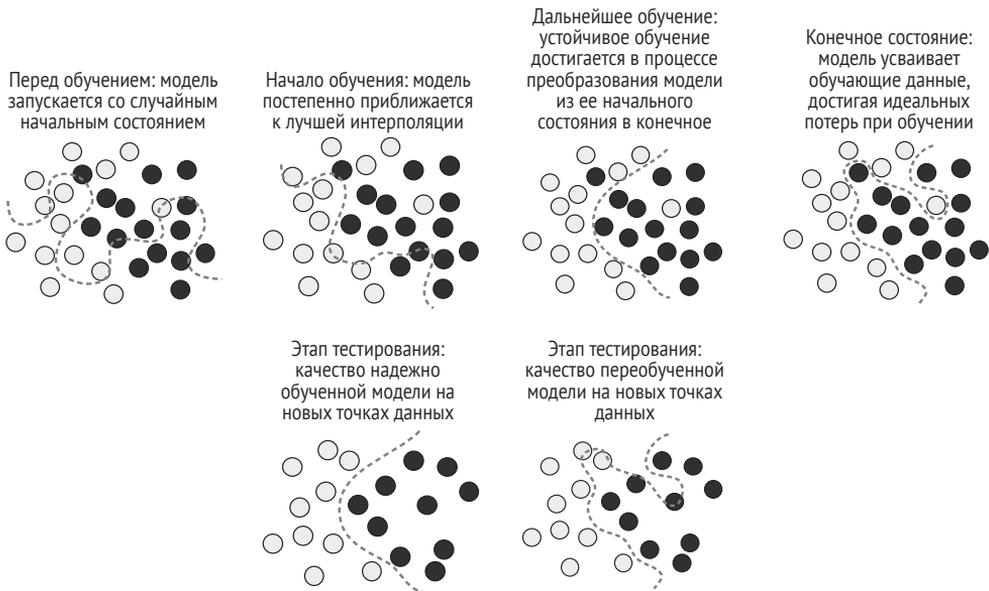


Рис. 5.10 Переход от случайной модели к модели с переобучением и достижение надежного обучения в качестве промежуточного состояния

Движение по кривой, изученной моделью в этой точке, будет близко к движению по фактическому скрытому многообразию данных. Следовательно, модель сможет понимать незнакомые входные данные посредством интерполяции между обучающими входными данными.

Помимо тривиального факта, что они обладают достаточной репрезентативной способностью, модели глубокого обучения обладают следующими свойствами, которые делают их особенно подходящими для изучения скрытых многообразий:

- они реализуют плавное непрерывное сопоставление входных данных с выходными. Оно должно быть гладким и непрерывным, потому что необходимо соблюдать условие дифференцируемости (иначе вы не могли бы использовать градиентный спуск). Эта гладкость помогает аппроксимировать скрытые многообразия, обладающие теми же свойствами;
- они, как правило, структурированы таким образом, чтобы отражать «форму» информации в обучающих данных (через априорно заданную архитектуру). Это особенно касается моделей

для обработки изображений (главы 8 и 9) и обработки последовательностей (глава 10). В более общем плане глубокие нейронные сети структурируют свои изученные представления иерархическим и модульным образом, что похоже на организацию естественных данных.

## Важность обучающих данных

Хотя глубокое обучение действительно хорошо подходит для изучения многообразий, способность к обобщению в первую очередь зависит от исходного состояния ваших данных, а не от конкретных свойств модели. Модель научится обобщать только в том случае, если ваши данные образуют многообразие, точки которого можно интерполировать. Чем более информативны и менее шумны признаки, тем лучше будет обобщать модель, потому что ваше пространство ввода будет проще и лучше структурировано. Но исходные данные обычно далеки от совершенства и плохо подходят для машинного обучения. Поэтому в машинном обучении существуют отдельные этапы *подготовки данных и конструирования признаков*.

Кроме того, поскольку глубокое обучение является подгонкой кривой, для того чтобы модель работала хорошо, *ее необходимо обучать на плотной выборке входного пространства*. Плотная выборка в данном случае означает, что обучающие данные должны плотно покрывать все множество входных данных (рис. 5.11). Это особенно важно вблизи границ принятия решений. При достаточно плотной выборке становится возможным осмыслить новые входные данные путем интерполяции между обучающими входными данными без необходимости использовать здравый смысл, абстрактные рассуждения или внешние знания о мире – все то, к чему модели машинного обучения не имеют доступа. Вы всегда должны помнить, что лучший способ усовершенствовать модель глубокого обучения – это обучить ее на большем количестве данных или на более качественных данных (очевидно, что добавление чрезмерно зашумленных или неточных данных повредит обобщению). Более плотное покрытие множества входных данных даст модель, которая лучше обобщает. Вы не можете ожидать, что модель глубокого обучения выполнит что-то большее, чем прямая интерполяция между своими обучающими выборками, и поэтому должны сделать все возможное, чтобы максимально упростить интерполяцию. Единственное, что вы можете получить от модели глубокого обучения, – это то, что вы в нее вложили: априорные данные, закодированные в ее архитектуре, и данные, на которых она обучалась.

Когда увеличение количества данных невозможно, следующим лучшим решением является сокращение объема информации, которую ваша модель может хранить, или добавление ограничений на гладкость моделируемой кривой. Если сеть может позволить себе запомнить лишь небольшое количество шаблонов, процесс оптими-

зации заставит ее сосредоточиться на наиболее заметных шаблонах, которые имеют больше шансов на хорошее обобщение. Этот способ борьбы с переобучением называется *регуляризацией*. Мы подробно рассмотрим методы регуляризации в разделе 5.4.4.



Рис. 5.11 Для обучения модели, способной к точному обобщению, необходима плотная выборка входного пространства

Прежде чем вы приступите к настройке своей модели для лучшего обобщения, вам понадобится способ оценить, как ваша модель работает в текущем состоянии. В следующем разделе вы узнаете, как отслеживать качество обобщения во время разработки модели.

## 5.2 Оценка моделей машинного обучения

Чтобы осмысленно управлять каким-либо объектом, нам нужно его наблюдать. Поскольку наша цель – разработать модели, способные успешно обобщать новые данные, очень важно иметь возможность надежно измерять способность модели к обобщению. В этом разделе мы подробно разберем различные способы оценки моделей машинного обучения. Вы уже видели большинство из них в действии в предыдущей главе.

### 5.2.1 Наборы данных для обучения, проверки и контроля

Оценка модели всегда сводится к разделению доступных данных на три набора: обучающий, проверочный и контрольный. Мы тщательно подгоняем модель к обучающим данным и оцениваем результат подгонки на проверочных данных. Как только модель будет готова к использованию на производстве, мы в последний раз тестируем

ее на контрольных данных, которые должны быть максимально похожи на производственные данные. Затем можно развернуть модель в рабочей среде.

Вы спросите, а почему бы не иметь всего два набора: обучающий и контрольный? Это гораздо проще!

Причина в том, что разработка модели всегда включает в себя настройку ее конфигурации, например выбор количества или размера слоев (называемых *гиперпараметрами* модели, чтобы отличать их от *параметров* – весов сети). Мы выполняем эту настройку, используя в качестве сигнала обратной связи качество модели на проверочных данных. По сути, настройка гиперпараметров является разновидностью обучения: поиском хорошей конфигурации в некотором пространстве параметров. В результате настройка конфигурации модели может быстро привести к переобучению на проверочном наборе, даже если модель никогда не обучалась на нем напрямую.

Центральное место в этом явлении занимает понятие *утечки информации*. Каждый раз, когда вы настраиваете гиперпараметр вашей модели на основе показателей модели на проверочном наборе, некоторая информация о проверочных данных просачивается в модель. Если вы сделаете это только один раз для одного параметра, то утечет очень мало информации, и ваш проверочный набор останется надежным средством оценки модели. Но если вы много раз повторите цикл обучения, проверки и настройки модели, в нее может просочиться значительный объем информации о проверочном наборе.

В конце концов, вы получите модель, которая искусственно хорошо работает на проверочных данных, потому что именно для этого вы ее оптимизировали. Но вам нужна модель, которая хорошо работает на совершенно новых данных, поэтому для окончательной оценки модели вам нужно использовать третий набор данных, т. е. контрольный. Ваша модель не должна иметь даже косвенного доступа к какой-либо информации о тестовом наборе. В противном случае мера обобщения вашей модели будет ошибочной.

Разделение ваших данных на обучающий, проверочный и контрольный наборы может показаться простым делом, но у нас есть несколько продвинутых способов, которые пригодятся, когда доступно мало данных. Давайте рассмотрим три классических рецепта оценки: простая проверка с отложенной выборкой,  $K$ -кратная проверка и повторная  $K$ -кратная проверка с перетасовкой. Мы также поговорим об использовании исходных данных, основанных на здравом смысле, для уверенности, что ваше обучение к чему-то приведет.

## ПРОСТАЯ ПРОВЕРКА С ОТЛОЖЕННОЙ ВЫБОРКОЙ

Отложите некоторую часть ваших данных в качестве контрольного набора. Обучите модель на оставшихся данных и оцените результат на контрольном наборе. Как было сказано в предыдущих разделах,

чтобы предотвратить утечку информации, вы не должны настраивать гиперпараметры модели на контрольном наборе и, следовательно, вам также следует отложить проверочный набор.

Схема данных для проверки с отложенной выборкой изображена на рис. 5.12. В листинге 5.5 показана простая реализация.

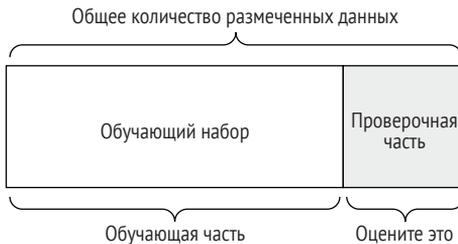


Рис. 5.12 Простая проверка с отложенной выборкой

### Листинг 5.5 Проверка с отложенной выборкой (для простоты метки опущены)

Обычно допустимо собирать контрольный набор из случайной выборки данных

```

num_validation_samples <- 10000
val_indices <- sample.int(num_validation_samples, nrow(data))
validation_data <- data[val_indices, ] ← Определяем контрольный набор
training_data <- data[-val_indices, ] ← Определяем тренировочный набор
model <- get_model()
fit(model, training_data, ...) ← Обучаем модель на обучающих данных
validation_score <- evaluate(model, validation_data, ...) ← и оцениваем ее на контрольных данных
...
model <- get_model()
fit(model, data, ...) ← На этом этапе вы можете настроить свою модель,
test_score <- evaluate(model, test_data, ...) ← обучить ее заново, оценить и снова настроить

```

После того как вы настроили свои гиперпараметры, окончательную модель обычно обучают с нуля на всех доступных данных (объединенные данные `training_data` и `validation_data`)

**ПРИМЕЧАНИЕ** В этих примерах мы предполагаем, что данные являются тензором второго ранга. Добавьте нужное количество запятых к вызову `[ ]` для данных более высокого ранга. Например, `data[idx, , ]` для данных третьего ранга, `data[idx, , , ]` для четвертого ранга и т. д.

Это самый простой протокол оценки, и он страдает одним недостатком: если доступно мало данных, ваши проверочные и контрольные наборы могут содержать слишком мало образцов, чтобы быть статистически репрезентативными для имеющихся данных.

Это легко распознать: если различные случайные раунды перетасовки данных перед разделением в конечном итоге дают очень разные показатели качества модели, то у вас возникла проблема нехватки данных.  $K$ -кратная проверка и повторная  $K$ -кратная проверка – два способа решить эту проблему. О них и пойдет речь дальше.

## К-КРАТНАЯ ПРОВЕРКА

Согласно этому протоколу оценки, разбейте свои данные на  $K$  разделов одинакового размера. Для каждого раздела  $i$  обучите модель на оставшихся  $K - 1$  разделах и оцените ее на разделе  $i$ . Ваша окончательная оценка – это среднее значение полученных  $K$  баллов. Этот метод полезен, когда показатель качества вашей модели показывает значительную дисперсию в зависимости от конкретного разделения данных на обучающий и контрольный наборы. Как и проверка с отложенной выборкой, этот метод не освобождает вас от использования отдельного проверочного набора для настройки модели.

Схематично  $K$ -кратная перекрестная проверка выглядит так, как показано на рис. 5.13 (повтор рис. 4.6 для наглядности). В листинге 5.6 представлена простая реализация.

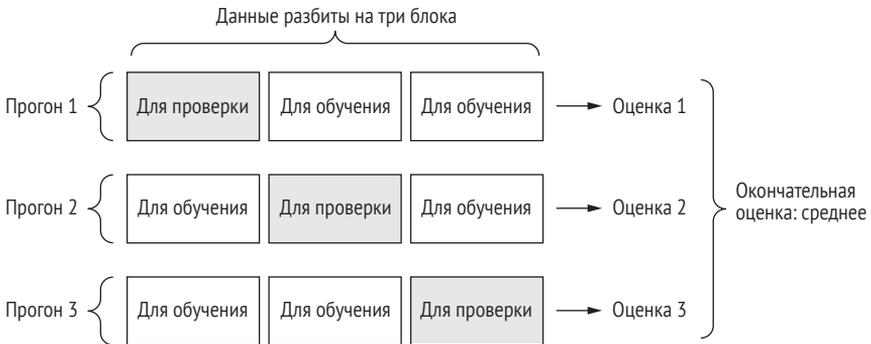


Рис. 5.13  $K$ -кратная перекрестная проверка с  $K = 3$

### Листинг 5.6 $K$ -кратная перекрестная проверка (для простоты метки опущены)

```
k <- 3
fold_id <- sample(rep(1:k, length.out = nrow(data)))
validation_scores <- numeric()

for (fold in seq_len(k)) {
  validation_idx <- which(fold_id == fold)
  validation_data <- data[validation_idx, ]
  training_data <- data[-validation_idx, ]
  model <- get_model()
  fit(model, training_data, ...)
}
```

Выбираем раздел контрольных данных

Используем остаток данных для обучения

Создаем новый экземпляр модели (необученный)

```

validation_score <- evaluate(model, validation_data, ...)
validation_scores[[fold]] <- validation_score
}
validation_score <- mean(validation_scores)
model <- get_model()
fit(model, data, ...)
test_score <- evaluate(model, test_data, ...)

```

Результат тестирования:  
среднее оценок K-fold

Обучаем окончательную модель на всех  
доступных не контрольных данных

## Повторная K-кратная проверка с перетасовкой

Этот протокол предназначен для ситуаций, в которых у вас относительно мало доступных данных и вам нужно как можно точнее оценить вашу модель. Я обнаружил, что он очень полезен в соревнованиях Kaggle. Методика заключается в многократном применении K-кратной проверки, с перемешиванием данных перед каждым разделением на K частей. Окончательная оценка представляет собой среднее значение оценок, полученных при каждом запуске K-кратной проверки. Обратите внимание, что в конечном итоге вы обучаете и оцениваете  $P * K$  моделей (где P – количество используемых вами итераций), что может быть очень дорого в вычислительном отношении.

## 5.2.2 Использование критериев, основанных на здравом смысле

Помимо различных протоколов оценивания, есть еще одна вещь, о которой вам следует знать, – это использование базовых критериев, основанных на здравом смысле.

Обучение модели похоже на нажатие кнопки, запускающей ракету в параллельном мире. Вы не можете ее услышать или увидеть. Вы не можете наблюдать процесс обучения на многообразии – он происходит в пространстве с тысячами измерений, и даже если вы спроецируете его в 3D, вы не сможете его интерпретировать. Единственная обратная связь, которую вы имеете, – это ваши метрики проверки, например высотомер на вашей ракете-невидимке.

Особенно важно уметь сказать, отрывается ли ваша ракета вообще от земли. На какую высоту она взлетела? Если ваша модель имеет точность 15 % – это хорошо? Прежде чем вы начнете работать с набором данных, вы всегда должны выбрать тривиальный базовый уровень, который намереваетесь превзойти. Если ваша модель преодолеет этот порог, вы будете знать, что находитесь на правильном пути: она действительно использует информацию из входных данных, чтобы делать обобщающие прогнозы, и вы можете продолжать работу. В качестве базового уровня можно выбрать точность случайного классификатора или простейшего метода, не имеющего от-

ношения к машинному обучению, который вы только можете себе представить.

Например, в примере цифровой классификации MNIST простым исходным уровнем будет точность проверки выше 0,1 (случайный классификатор); в примере с IMDB мы ожидаем точность проверки выше 0,5. В примере со статьями Reuters это будет около 0,18–0,19 из-за дисбаланса классов. Если у вас есть проблема с бинарной классификацией, когда 90 % выборок принадлежат классу А и лишь 10 % принадлежат классу В, то классификатор, который всегда предсказывает А, естественным образом достигает точности 0,9 на проверочных данных, и вам нужно добиться лучшего результата.

Наличие здравого смысла, на который вы можете опираться, очень важно, когда вы приступаете к задаче, которую никто раньше не решал. Если вы не можете победить тривиальное решение, ваша модель бесполезна – возможно, вы используете неправильную модель или задача, над которой вы работаете, вообще не может быть решена с помощью машинного обучения. Пора вернуться к чертежной доске.

### 5.2.3 Что следует помнить об оценке модели

При выборе протокола оценки обратите внимание на следующие аспекты:

- *репрезентативность данных* – необходимо, чтобы и обучающий, и тестовый наборы были репрезентативными для имеющихся данных. Например, если вы пытаетесь классифицировать изображения цифр и начинаете с массива образцов, упорядоченных по их классам, принимая первые 80 % массива в качестве обучающего набора, а оставшиеся 20 % оставляете для контроля, это приведет к тому, что ваш обучающий набор будет содержать только классы 0–7, тогда как контрольный набор будет содержать лишь классы 8–9. Это очевидная нелепая ошибка, но она встречается на удивление часто. По этой причине принято случайным образом перемешивать данные, прежде чем разбивать их на обучающие и контрольные наборы;
- *ось времени* – если вы пытаетесь предсказать будущее на основе прошлого (например, завтрашняя погода, движение акций и т. д.), вам не следует случайным образом перемешивать данные перед их разделением, потому что это создаст *утечку во времени*: ваша модель будет усердно обучаться на данных из будущего. В таких ситуациях вы всегда должны убедиться, что все данные в вашем контрольном наборе находятся после данных в обучающем наборе на оси времени;
- *избыточность данных*. Если некоторые точки данных в вашем наборе появляются дважды (что довольно часто встречается в реальных данных), то перетасовка данных и разделение их на обучающий и проверочный наборы приведут к формированию пересекающихся наборов. По сути, вы будете частично тестиро-

вать модель на обучающих данных, а это худшее, что вы можете сделать! Убедитесь, что наборы не пересекаются.

Имея надежный способ оценки точности вашей модели, вы сможете отслеживать вечную гонку за компромиссами, лежащими в основе машинного обучения – между оптимизацией и обобщением, недообучением и переобучением.

## 5.3 Улучшение качества обучения модели

Чтобы добиться идеального обучения модели, ее сначала нужно переобучить. Если вы не знаете заранее, где лежит граница, вам придется пересечь ее, чтобы найти. Следовательно, ваша первоначальная цель, когда вы начинаете работать над задачей, состоит в том, чтобы получить модель, которая хоть в какой-то степени обобщает и способна к переобучению. Далее вы начинаете улучшать обобщение, попутно борясь с переобучением.

На этом этапе вы столкнетесь с тремя распространенными проблемами:

- обучение не начинается – потери на обучающем наборе не уменьшаются со временем;
- обучение начинается очень хорошо, но ваша модель не дает осмысленного обобщения – вы не можете превзойти установленный вами базовый уровень, основанный на здравом смысле;
- потери при обучении и проверке со временем снижаются, и вы можете превзойти базовый уровень, но пока не можете добиться переобучения, т. е. ваша модель все еще недостаточно обучена.

Давайте посмотрим, как вы можете решить эти проблемы, чтобы достичь первой важной вехи в проекте машинного обучения: получить модель, которая обладает некоторой способностью к обобщению (может превзойти базовый уровень) и способна к переобучению.

### 5.3.1 Настройка ключевых параметров градиентного спуска

Иногда обучение не начинается или останавливается слишком рано. Ваша функция потерь застряла. Эту проблему *всегда* можно решить: помните, что вы можете подогнать модель даже к случайным данным. Даже если ваша задача абсолютно не имеет смысла, вы все равно должны быть в состоянии обучить модель – хотя бы путем запоминания обучающих данных.

Неполадки с процессом обучения всегда говорят о наличии той или иной ошибки в параметрах градиентного спуска: выбор оптимизатора, распределение начальных значений весов модели, ско-

рость обучения или размер пакета данных. Все они взаимосвязаны, и поэтому обычно достаточно настроить скорость обучения и размер партии, не меняя остальные параметры.

Давайте рассмотрим конкретный пример: обучение модели MNIST из главы 2 с неадекватно большой скоростью обучения, равной 1.

### Листинг 5.7 Обучение модели MNIST с неадекватно большой скоростью

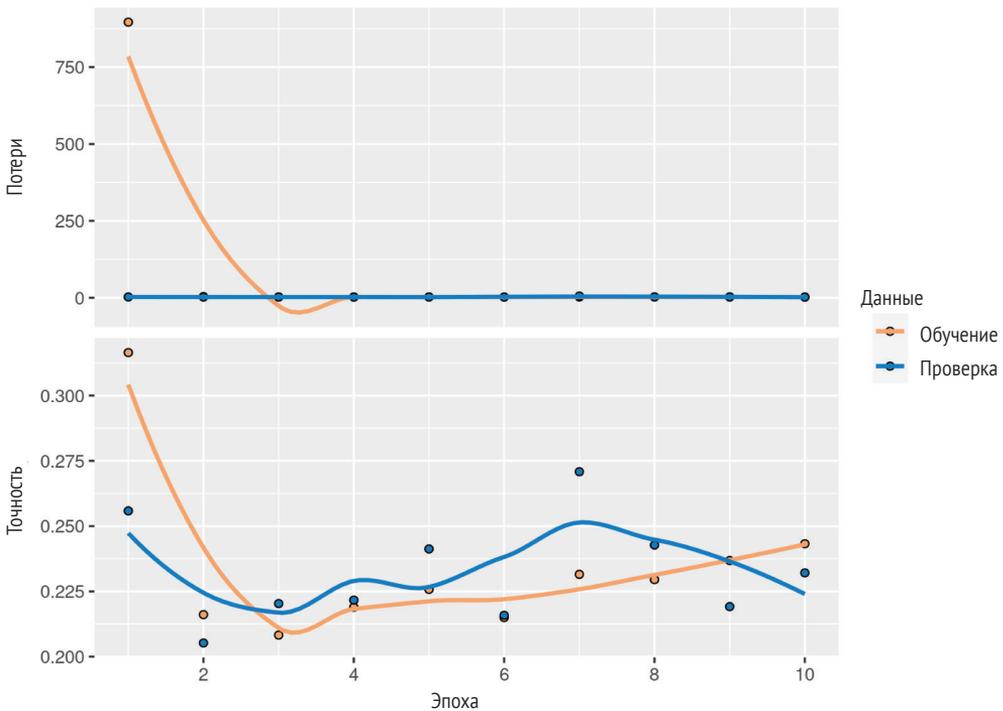
```
c(c(train_images, train_labels), .) %<- dataset_mnist()
train_images <- array_reshape(train_images / 255,
                             c(60000, 28 * 28))

model <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

model %>% compile(optimizer = optimizer_rmsprop(1),
                 loss = "sparse_categorical_crossentropy",
                 metrics = "accuracy")

history <- model %>% fit(train_images, train_labels,
                        epochs = 10, batch_size = 128,
                        validation_split = 0.2)

plot(history)
```



Модель быстро достигает точности обучения и проверки в диапазоне 20–30 %, но не может выйти за ее пределы. Давайте попробуем снизить скорость обучения до более разумного значения  $1e-2$ .

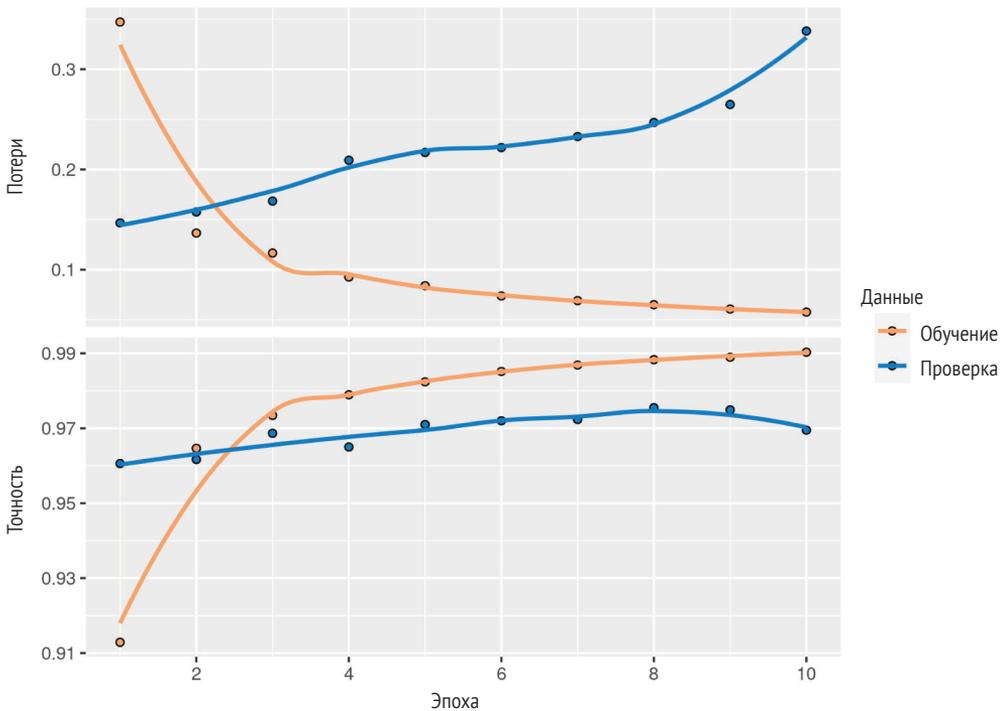
#### Листинг 5.8 Та же модель с более подходящей скоростью обучения

```
model <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

model %>% compile(optimizer = optimizer_rmsprop(1e-2),
  loss = "sparse_categorical_crossentropy",
  metrics = "accuracy")

model %>%
  fit(train_images, train_labels,
    epochs = 10, batch_size = 128,
    validation_split = 0.2) ->
  history

plot(history)
```



Теперь модель нормально обучается.

Если вы оказались в похожей ситуации, попробуйте следующие решения:

- понижение или увеличение скорости обучения. Слишком высокая скорость обучения может привести к обновлениям, которые значительно превышают правильную подгонку, как в предыдущем примере, а слишком низкая скорость обучения может сделать обучение настолько медленным, что кажется, будто оно останавливается;
- увеличение размера партии – партия с большим количеством выборок приведет к более информативным и менее шумным градиентам (более низкая дисперсия).

В конце концов вы найдете конфигурацию, с которой начнется обучение.

### 5.3.2 *Использование лучшей априорно обоснованной архитектуры*

У вас есть обученная модель, но по какой-то причине ее точность на проверочных данных совсем не улучшилась. Она осталась на уровне случайного классификатора: ваша модель обучается, но не обобщает. В чем дело?

Это, возможно, худшая ситуация с машинным обучением, в которой вы можете оказаться. Она означает, что у вас *принципиально неправильный подход к решению задачи*, и может быть весьма непросто выяснить, в чем именно вы ошиблись. Вот несколько советов.

Во-первых, может случиться так, что используемые вами входные данные просто не содержат достаточной информации для предсказания ваших целей: проблема в том виде, в каком она сформулирована, неразрешима. Мы сталкивались с этим ранее, когда пытались обучить модель MNIST на данных с перемешанными метками: модель прекрасно обучалась, но точность на проверочном наборе оставалась на уровне 10 %, потому что с таким набором данных было просто невозможно вывести обобщение.

Во-вторых, может случиться так, что используемая вами модель не подходит для решения поставленной задачи. Например, в главе 10 вы увидите пример задачи прогнозирования временных рядов, когда архитектура с полносвязными слоями не может превзойти тривиальный исходный уровень, в то время как более подходящая рекуррентная архитектура хорошо находит обобщение. Использование модели, в которую изначально заложены правильные предположения о проблеме, важно для достижения обобщения: вы должны использовать правильные априорные архитектурные решения.

В следующих главах вы узнаете о наиболее подходящих архитектурах, которые можно использовать для различных модальностей данных – изображений, текста, временных рядов и т. д. В общем, вы всегда должны ознакомиться с рекомендациями по архитектуре для

того типа задачи, за которую беретесь – скорее всего, вы не первый, кто пытается это сделать.

### 5.3.3 Увеличение емкости модели

Если вам удалось найти подходящую модель, у которой точность растет по мере обучения и наблюдаются хоть какие-то способности к обобщению, вас можно поздравить: вы почти у цели. Теперь вам нужно заставить вашу модель достичь переобучения. Рассмотрим следующую небольшую модель – простую логистическую регрессию, – обученную на пикселях MNIST.

#### Листинг 5.9 Простая логистическая регрессия, обученная на данных MNIST

```
model <- keras_model_sequential() %>%  
  layer_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)  
  
plot(history_small_model$metrics$val_loss, type = 'o',  
  main = "Effect of Insufficient Model Capacity on Validation Loss",  
  xlab = "Epochs", ylab = "Validation Loss")
```

Вы получаете кривые потерь, которые выглядят как на рис. 5.14.

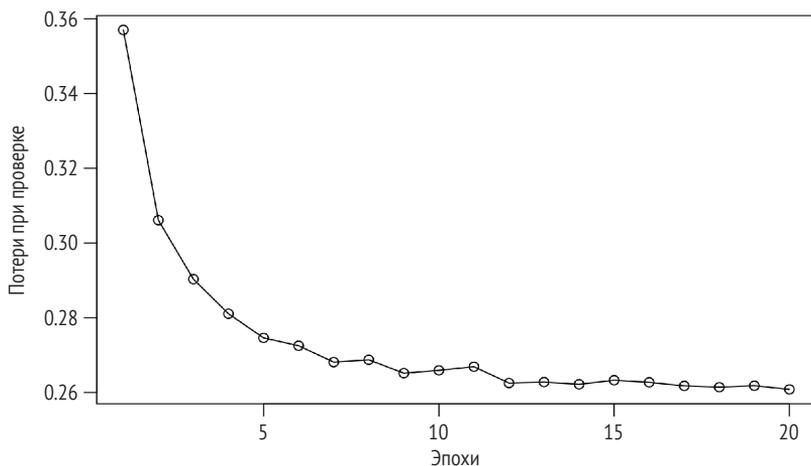


Рис. 5.14 Влияние недостаточной емкости модели на потери при проверке

Метрика потерь на проверочном наборе, кажется, останавливается или улучшается очень медленно, вместо того чтобы достигать пика и менять направление. Потери при проверке доходят до 0,26 и просто остаются на этом уровне. Вы никак не можете достичь явного переобучения даже после многих итераций по обучающим данным. Наверняка вы часто будете сталкиваться с подобными кривыми в своей деятельности.

Помните, что *всегда* должна быть возможность переобучения. И это тоже проблема, которую всегда можно решить. Если не удастся достичь стадии переобучения, скорее всего, проблема заключается в низкой *репрезентативной мощности* модели: вам понадобится модель большего размера, с большей *емкостью*, то есть способная хранить больше информации. Вы можете увеличить репрезентативную мощность, добавив больше слоев, используя более крупные слои (слои с большим количеством параметров) или типы слоев, которые больше подходят для рассматриваемой проблемы (лучшая априорная архитектура).

Давайте попробуем обучить более крупную модель с двумя промежуточными слоями по 96 единиц в каждом:

```
model <- keras_model_sequential() %>%
  layer_dense(96, activation = "relu") %>%
  layer_dense(96, activation = "relu") %>%
  layer_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 именно так, как должна: модель быстро обучается и начинает переобучаться после восьми эпох (рис. 5.15):

```
plot(history_large_model$metrics$val_loss, type = 'o',
      main = "Validation Loss for a Model with Appropriate Capacity",
      xlab = "Epochs", ylab = "Validation Loss")
```

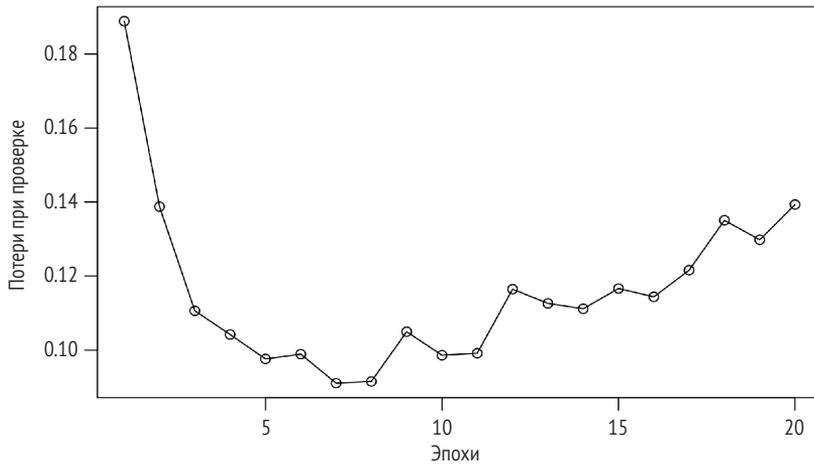


Рис. 5.15 Потери на проверочных данных для модели с достаточной емкостью

## 5.4 Как улучшить обобщение

Как только ваша модель показала, что обладает некоторой способностью к обобщению и способна переобучаться, пришло время переключить ваше внимание на достижение наилучшего обобщения.

### 5.4.1 Подготовка набора данных

Вы уже знаете, что обобщение в глубоком обучении основано на выявлении скрытой структуры ваших данных. Если ваши данные позволяют плавно интерполировать выборки, вы сможете обучить модель глубокого обучения, которая способна обобщать. Если ваши данные чрезмерно зашумлены или задача принципиально дискретная, как, скажем, сортировка списка, глубокое обучение вам не поможет. Глубокое обучение – это подгонка непрерывной кривой к данным, а не магия.

Именно поэтому важно убедиться, что вы работаете с подходящим набором данных. Инвестиции в сбор и подготовку хорошего набора данных всегда выгоднее, чем затраты на разработку лучшей модели.

- Убедитесь, что у вас достаточно данных. Помните, что вам нужна *плотная выборка* пространства ввода-вывода. Чем больше данных, тем лучше модель. Иногда задачи, которые сначала кажутся неразрешимыми, поддаются с большим набором данных.
- Сведите к минимуму ошибки аннотирования – визуализируйте свои входные данные, чтобы проверить наличие аномалий, и проверьте сомнительные метки.
- Очистите свои данные и разберитесь с отсутствующими значениями (об этом пойдет речь в следующей главе).
- Если у вас много признаков и вы не уверены, что все они действительно полезны, займитесь *отбором признаков*.

Особенно важным способом повышения обобщающего потенциала ваших данных является конструирование признаков. Этот прием выступает ключевой составляющей успеха для большинства задач машинного обучения. Давайте рассмотрим его подробнее.

## 5.4.2 Конструирование признаков

*Конструирование признаков* – это процесс использования ваших собственных знаний о данных и имеющемся алгоритме машинного обучения (в данном случае – нейронной сети), чтобы алгоритм работал лучше. Смысл в том, чтобы применить жестко закодированные (необучаемые) преобразования к данным до того, как они поступят в модель. Во многих случаях неразумно ожидать, что модель машинного обучения сможет обучиться на совершенно произвольных данных. Состав и форма данных должны быть подобраны так, чтобы облегчить работу модели.

Давайте рассмотрим интуитивно понятный пример. Предположим, вы пытаетесь разработать модель, которая получает в качестве входных данных изображение стрелочных часов и выводит время суток (рис. 5.16).



**Рис. 5.16** Конструирование признаков для чтения времени на стрелочных часах

Если вы решите напрямую использовать необработанные пиксели изображения в качестве входных данных, перед вами встанет сложная задача машинного обучения, связанная с компьютерным зрением. Для ее решения вам понадобится сверточная нейронная сеть, и вам придется потратить немало вычислительных ресурсов на обучение сети.

Но если вы уже понимаете суть задачи на высоком уровне (вы понимаете, как люди читают время на циферблате), вы можете придумать гораздо лучшие входные признаки для алгоритма машинного обучения: например, не составит особого труда написать пятистрочный R-скрипт для отслеживания черных пикселей стрелок часов и вывода координат  $(x, y)$  кончика каждой стрелки. Затем простой алгоритм машинного обучения может научиться связывать эти координаты с временем суток.

Вы можете пойти еще дальше: изменить систему координат и выразить координаты  $(x, y)$  как полярные координаты относительно центра изображения. Ваши данные превратятся в угол  $\theta$  для каждой стрелки часов. Теперь новые признаки настолько упрощают задачу, что можно обойтись без машинного обучения; простой операции округления и поиска в словаре достаточно, чтобы с приемлемой точностью установить время суток.

В этом суть конструирования признаков: упростить задачу, выражая ее более простым способом. Сделайте скрытое многообразие более гладким, простым и лучше организованным. Для этого обычно требуется глубокое понимание задачи.

До глубокого обучения конструирование признаков было наиболее важной частью рабочего процесса машинного обучения, потому что классические неглубокие алгоритмы не имели достаточно обширного пространства гипотез, чтобы самостоятельно изучать полезные признаки. Удачное представление данных алгоритму было абсолютно необходимым компонентом успеха. Например, до того, как сверточные нейронные сети научились успешно классифицировать цифры MNIST, решения обычно основывались на жестко запрограммированных признаках, таких как количество замкнутых петель в изображении цифры, высота каждой цифры в изображении, гистограмма значений пикселей и т. д. К счастью, современное глубокое обучение устраняет необходимость в конструировании большинства признаков, поскольку нейронные сети способны автоматически извлекать полезные признаки из необработанных данных. Означает ли это, что вам не нужно беспокоиться о конструировании признаков, когда вы используете глубокие нейронные сети? Нет, по следующим двум причинам:

- хорошие признаки по-прежнему позволяют решать проблемы более элегантно, используя меньше ресурсов. Например, было бы нелепо решать задачу чтения циферблата стрелочных часов с помощью сверточной нейронной сети;

- хорошие признаки позволяют решить проблему с гораздо меньшим объемом данных. Способность моделей глубокого обучения самостоятельно изучать признаки напрямую зависит от наличия большого количества доступных обучающих данных; если у вас всего несколько образцов, информативность их признаков становится критически низкой.

### 5.4.3 Использование ранней остановки

В глубоком обучении мы всегда используем модели, которые обладают огромным излишком параметров: у них гораздо больше степеней свободы, чем минимум, необходимый для подгонки к скрытому многообразию данных. Эта чрезмерная параметризация не является проблемой, потому что *вы никогда не обучаете модель до конца*. При обучении «до упора» модель вообще не будет обобщать. Вы всегда будете прерывать обучение задолго до того, как достигнете минимально возможных потерь на обучающих данных.

Поиск точки на кривой потерь, в которой модель достигает наилучшего обобщения – точной границы между недостаточным обучением и переобучением, – является одним из самых эффективных способов улучшить обобщение.

В примерах из предыдущей главы мы начали с обучения наших моделей дольше, чем необходимо, чтобы определить количество эпох, которые дают наилучшие значения метрик на проверочном наборе, а затем могли бы переобучить новую модель именно для этого количества эпох. Это довольно стандартный подход, но он требует выполнения избыточных вычислений, которые иногда обходятся очень дорого. Естественно, вы можете просто сохранять свою модель в конце каждой эпохи, и как только вы найдете лучшую эпоху, начать использовать соответствующую сохраненную модель. В Keras это обычно делается с помощью метода `callback_early_stopping`, который прерывает обучение, как только метрики модели на проверочном наборе перестают улучшаться, при этом запоминая последнее наилучшее состояние модели. Вы научитесь использовать этот метод в главе 7.

### 5.4.4 Регуляризация модели

*Методы регуляризации* – это набор специальных приемов, которые активно препятствуют склонности модели к переобучению с целью повышения точности модели на проверочных данных. Регуляризация получила такое название, потому что она стремится сделать модель более простой, более «регулярной», а ее кривую – более плавной, более «общей»; таким образом, модель становится менее специфичной для обучающего набора и лучше подходит для обобщения за счет более точного приближения к скрытому многообразию данных.

Имейте в виду, что регуляризация модели – это процесс, который всегда зависит от точности процедуры оценки. Вы добьетесь хорошего обобщения, только если сможете его измерить.

Далее мы рассмотрим некоторые из наиболее распространенных методов регуляризации и применим их на практике для улучшения модели классификации отзывов о фильмах из главы 4.

## УМЕНЬШЕНИЕ РАЗМЕРА СЕТИ

Вы уже знаете, что слишком маленькая модель не будет переобучаться. Самый простой способ уменьшить склонность к переобучению – уменьшить размер модели (количество обучаемых параметров в модели, определяемое количеством слоев и количеством единиц на уровне). Если модель имеет ограниченные ресурсы запоминания, она не сможет просто запомнить свои обучающие данные; таким образом, чтобы свести к минимуму свои потери, ей придется прибегнуть к изучению сжатых представлений, обладающих предсказательной силой в отношении целей, – это именно тот тип представлений, который нас интересует. В то же время имейте в виду, что вы должны использовать модели, которые имеют достаточно параметров – ваша модель не должна испытывать нехватку ресурсов для запоминания. Необходимо найти компромисс между *слишком большой емкостью* и *недостаточной емкостью*.

К сожалению, не существует волшебной формулы для определения правильного количества слоев или правильного размера каждого слоя. Вам придется оценить несколько различных архитектур (разумеется, на проверочном, а не на контрольном наборе), чтобы найти правильный размер модели для ваших данных. Стандартный рабочий процесс для поиска подходящего размера модели заключается в том, чтобы начать с относительно небольшого количества слоев и параметров и увеличивать размер слоев или добавлять новые слои до тех пор, пока вы не увидите уменьшение потерь на проверочном наборе.

Давайте опробуем этот подход на модели классификации кинообзоров. В листинге 5.10 показана наша исходная модель.

### Листинг 5.10 Исходная модель

```
c(c(train_data, train_labels), .) %<-% dataset_imdb(num_words = 10000)

vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for(i in seq_along(sequences))
    results[i, sequences[[i]]] <- 1
  results
}

train_data <- vectorize_sequences(train_data)
```

```

model <- keras_model_sequential() %>%
  layer_dense(16, activation = "relu") %>%
  layer_dense(16, activation = "relu") %>%
  layer_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 давайте попробуем заменить ее на уменьшенную модель (листинг 5.11).

#### Листинг 5.11 Версия модели с уменьшенной емкостью

```

model <- keras_model_sequential() %>%
  layer_dense(4, activation = "relu") %>%
  layer_dense(4, activation = "relu") %>%
  layer_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)

```

Построим график потерь (рис. 5.17), чтобы сравнить потери исходной и уменьшенной моделей:

```

plot(
  NULL,
  main = "Original Model vs. Smaller Model on IMDB Review
Classification",
  xlab = "Эпохи",
  xlim = c(1, history_original$params$epochs),
  ylab = "Потери",
  ylim = extendrange(history_original$metrics$val_loss),
  panel.first = abline(v = 1:history_original$params$epochs,
                       lty = "dotted", col = "lightgrey")
)

lines(history_original $metrics$val_loss, lty = 2)
lines(history_smaller_model$metrics$val_loss, lty = 1)
legend("topleft", lty = 2:1,
      legend = c("Потери при тестировании исходной модели",
                 "Потери при тестировании меньшей модели"))

```

NULL указывает команде plot() построить область вывода, но не выводить данные

Рисование линий сетки

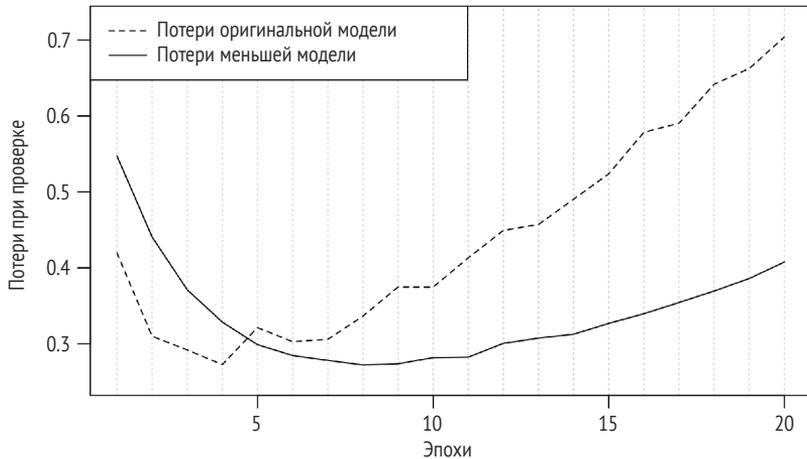


Рис. 5.17 Сравнение потерь исходной и уменьшенной моделей на данных классификации обзоров IMDB

Как видно на графике, уменьшенная модель начинает переобучаться позже, чем исходная (после шести эпох, а не четырех), и ее точность снижается медленнее с момента начала переобучения.

Теперь давайте возьмем модель, которая имеет увеличенную емкость – гораздо большую, чем требует задача. Хотя обычно принято работать с моделями, параметров которых слишком много для изучаемых данных, объективно существует такое явление, как слишком большая способность к запоминанию. Вы поймете, что ваша модель слишком велика, если она очень быстро достигнет переобучения и если ее кривая потерь на проверочных данных выглядит ломаной линией с высокой дисперсией (хотя изменчивый характер кривой потерь также может быть признаком использования ненадежной методики проверки).

#### Листинг 5.12 Версия модели с увеличенной емкостью

```
model <- keras_model_sequential() %>%
  layer_dense(512, activation = "relu") %>%
  layer_dense(512, activation = "relu") %>%
  layer_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)

plot(
  NULL,
```

```

main =
  "Original Model vs. Much Larger Model on IMDB Review Classification",
xlab = "Эпохи", xlim = c(1, history_original$params$epochs),
ylab = "Потери",
ylim = range(c(history_original$metrics$val_loss,
               history_larger_model$metrics$val_loss)),
panel.first = abline(v = 1:history_original$params$epochs,
                    lty = "dotted", col = "lightgrey")
)
lines(history_original $metrics$val_loss, lty = 2)
lines(history_larger_model$metrics$val_loss, lty = 1)
legend("topleft", lty = 2:1,
      legend = c("Потери при тестировании исходной модели",
                 "Потери при тестировании большей модели"))

```

На рис. 5.18 показано, как работает увеличенная модель по сравнению с исходной.

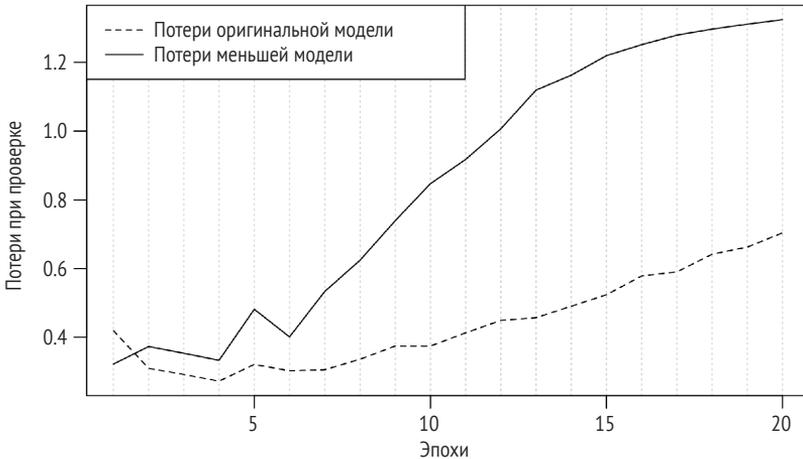


Рис. 5.18 Сравнение потерь исходной и увеличенной моделей на данных классификации обзоров IMDB

Увеличенная модель начинает переобучаться почти сразу, всего через одну эпоху, и переобучение выражено гораздо сильнее. Ее кривая потерь на проверочных данных также более шумная. При этом она очень быстро приближается к нулю на обучающих данных. Чем больше объем модели, тем быстрее она может смоделировать обучающие данные (что приводит к низким потерям при обучении), но тем больше она подвержена переобучению (что приводит к большой разнице между потерями при обучении и проверке).

## ДОБАВЛЕНИЕ РЕГУЛЯРИЗАЦИИ ВЕСА

Возможно, вы знакомы с принципом *бритвы Оккама*: при наличии двух объяснений чего-либо наиболее верным объяснением, скорее

всего, будет самое простое – то, в котором делается меньше предположений. Эта идея также применима к моделям на основе обученных нейронных сетей: при заданной сетевой архитектуре один и тот же набор данных можно представить с помощью разных наборов весовых коэффициентов (разных моделей). Более простые модели менее склонны к переобучению, чем сложные.

*Простая модель* в этом контексте – это модель, в которой распределение значений параметров имеет меньшую энтропию (или модель с меньшим количеством параметров, как вы видели в предыдущем разделе). Поэтому распространенный способ борьбы с переобучением состоит в том, чтобы наложить ограничения на сложность модели, заставив ее веса принимать только небольшие значения, что делает распределение значений веса более равномерным (регулярным). Этот прием называется *регуляризацией веса* и реализуется путем прибавления к функции потерь стоимости, связанной с наличием больших весов. Эта стоимость бывает двух видов:

- *регуляризация L1* – прибавляемая стоимость пропорциональна абсолютному значению весовых коэффициентов (норма весов L1);
- *регуляризация L2* – прибавляемая стоимость пропорциональна квадрату значения весовых коэффициентов (норма весов L2). Регуляризация L2 также называется *уменьшением веса* (weight decay) в контексте нейронных сетей. Не позволяйте другому названию сбить вас с толку: термин «уменьшение веса» математически означает то же самое, что и «регуляризация L2».

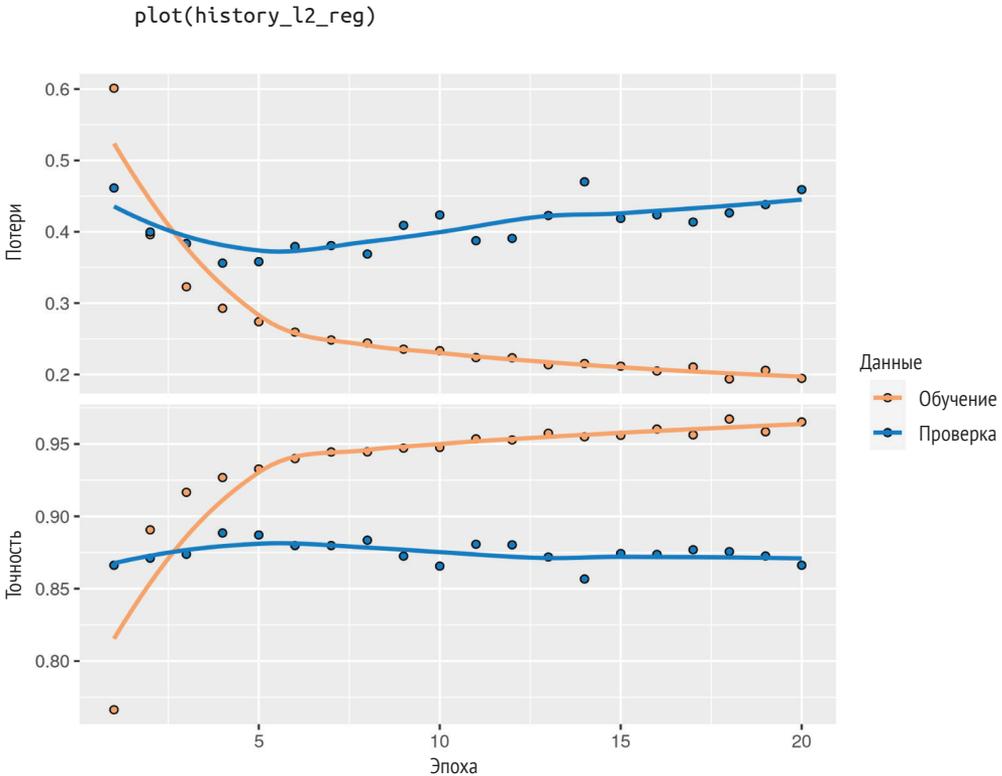
В Keras регуляризация веса добавляется в модель путем передачи *экземпляров регуляризатора веса* слоям в качестве аргументов ключевого слова. Давайте добавим регуляризацию веса L2 к нашей исходной модели классификации обзоров фильмов.

### Листинг 5.13 Добавление в модель весовой регуляризации L2

```
model <- keras_model_sequential() %>%
  layer_dense(16, activation = "relu",
              kernel_regularizer = regularizer_l2(0.002)) %>%
  layer_dense(16, activation = "relu",
              kernel_regularizer = regularizer_l2(0.002)) %>%
  layer_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)
```



В предыдущем листинге `regularizer_l2(0.002)` означает, что каждый коэффициент в матрице весов слоя будет добавлять  $0.002 * \text{weight\_coefficient\_value}^2$  к общим потерям модели. Обратите внимание, что поскольку этот штраф добавляется только во время обучения, потери для этой модели будут намного выше во время обучения, чем во время тестирования.

На рис. 5.19 показано влияние штрафа за регуляризацию L2. Как видите, модель с регуляризацией L2 стала гораздо более устойчивой к переобучению, чем эталонная модель, несмотря на то что обе модели имеют одинаковое количество параметров.

#### Листинг 5.14 Построение графика для демонстрации эффекта регуляризации веса L2

```
plot(NULL,
      main = "Effect of L2 Weight Regularization on Validation Loss",
      xlab = "Эпохи", xlim = c(1, history_original$params$epochs),
      ylab = "Потери",
      ylim = range(c(history_original$metrics$val_loss,
                    history_l2_reg $metrics$val_loss)),
      panel.first = abline(v = 1:history_original$params$epochs,
                          lty = "dotted", col = "lightgrey"))
lines(history_original$metrics$val_loss, lty = 2)
```

```
lines(history_l2_reg $metrics$val_loss, lty = 1)
legend("topleft", lty = 2:1,
      legend = c("Потери при тестировании исходной модели",
                 "Потери при тестировании большей модели"))
```

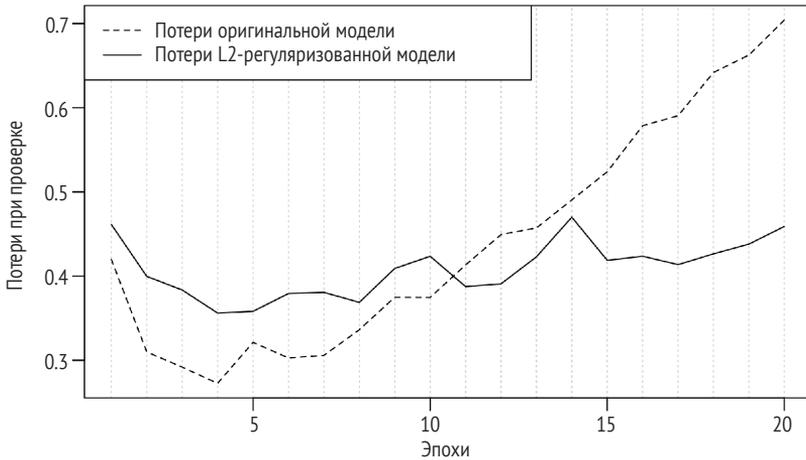


Рис. 5.19 Влияние регуляризации веса L2 на потери при проверке

В качестве альтернативы регуляризации L2 вы можете использовать один из следующих регуляризаторов веса Keras.

**Листинг 5.15** Различные регуляризаторы веса, доступные в Keras

```
regularizer_l1(0.001) ← Регуляризация L1
regularizer_l1_l2(l1 = 0.001, l2 = 0.001) ← Одновременно
                                             регуляризация L1 и L2
|
| <keras.regularizers.L1 object at 0x7f81cc3df340>
| <keras.regularizers.L1L2 object at 0x7f81cc651c40>
```

Имейте в виду, что регуляризация веса чаще используется для небольших моделей глубокого обучения. Большие модели глубокого обучения, как правило, настолько чрезмерно параметризованы, что наложение ограничений на значения весов не оказывает большого влияния на емкость и обобщение модели. В этих случаях предпочтительнее другой метод регуляризации: прореживание.

**ДОБАВЛЕНИЕ ПРОРЕЖИВАНИЯ**

*Прореживание (dropout)* – один из самых эффективных и наиболее часто используемых методов регуляризации нейронных сетей. Он разработан Джеффом Хинтоном и его студентами из Университета Торонто. Прореживание, применяемое к слою, состоит в случайном исключении (обнулении) ряда выходных признаков слоя во время обучения. Допустим, некий слой обычно возвращает вектор  $s(0.2, 0.5, 1.3, 0.8, 1.1)$  для данной входной выборки во время обуче-

ния. После применения прореживания в этом векторе будет несколько нулевых элементов, распределенных случайным образом: например,  $c(0, 0.5, 1.3, 0, 1.1)$ . Коэффициент прореживания – это доля признаков, которые обнулены; обычно он принимает значение между 0,2 и 0,5. Во время тестирования ни один элемент не отбрасывают; вместо этого выходные значения слоя уменьшаются на коэффициент, равный коэффициенту прореживания, чтобы сбалансировать тот факт, что активных элементов больше, чем во время обучения.

Рассмотрим матрицу, содержащую выходные данные слоя `layer_output` с формой `(batch_size, features)`. Во время обучения мы случайным образом обнуляем часть значений в матрице:

```
zero_out <- random_array(dim(layer_output)) < .5
layer_output[zero_out] <- 0
```

← Во время обучения обнуляем  
50 % значений

Во время тестирования мы уменьшаем вывод на коэффициент прореживания. В данном случае умножаем на 0,5 (потому что ранее мы обнулили половину элементов):

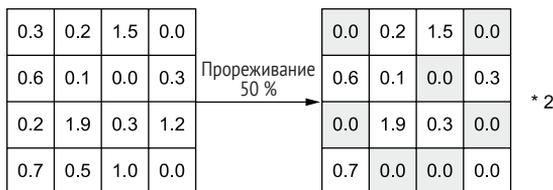
```
layer_output <- layer_output * .5
```

← Во время тестирования

Заметим, что этот процесс можно реализовать, выполняя обе операции во время обучения и оставляя выходные данные неизменными во время тестирования, как часто и происходит на практике (рис. 5.20):

```
layer_output[random_array(dim(layer_output)) < dropout_rate] <- 0
layer_output <- layer_output / .5
```

← Обучение. Обратите внимание,  
что в данном случае мы увеличиваем,  
а не уменьшаем масштаб



**Рис. 5.20** Прореживание матрицы активации и последующее масштабирование происходят во время обучения. Во время тестирования матрица активации не изменяется

Этот прием может показаться странным и взятым с потолка. Почему он должен уменьшить переобучение? Хинтон говорит, что среди прочего его натолкнул на эту идею механизм предотвращения мошенничества, используемый банками. По его словам, это было так: «Я регулярно ходил в отделение своего банка. Сотрудники отделения постоянно менялись, и я спросил одного из них, почему так происходит. Он ответил, что не знает, но сотрудников и в самом деле часто чередовали. Я подумал, что таким образом устраняют возможность сговора между сотрудниками с целью обмана банка. Это навело меня на мысль, что случайное удаление разных подмножеств нейронов в каждом проходе предотвратит заговоры и, таким образом, уменьшит переобучение». Основная идея заключается в том, что введение шума в выходные значения слоя может разрушить несущественные случайные шаблоны (то, что Хинтон называет «заговорами»), которые модель могла бы запомнить в отсутствие шума.

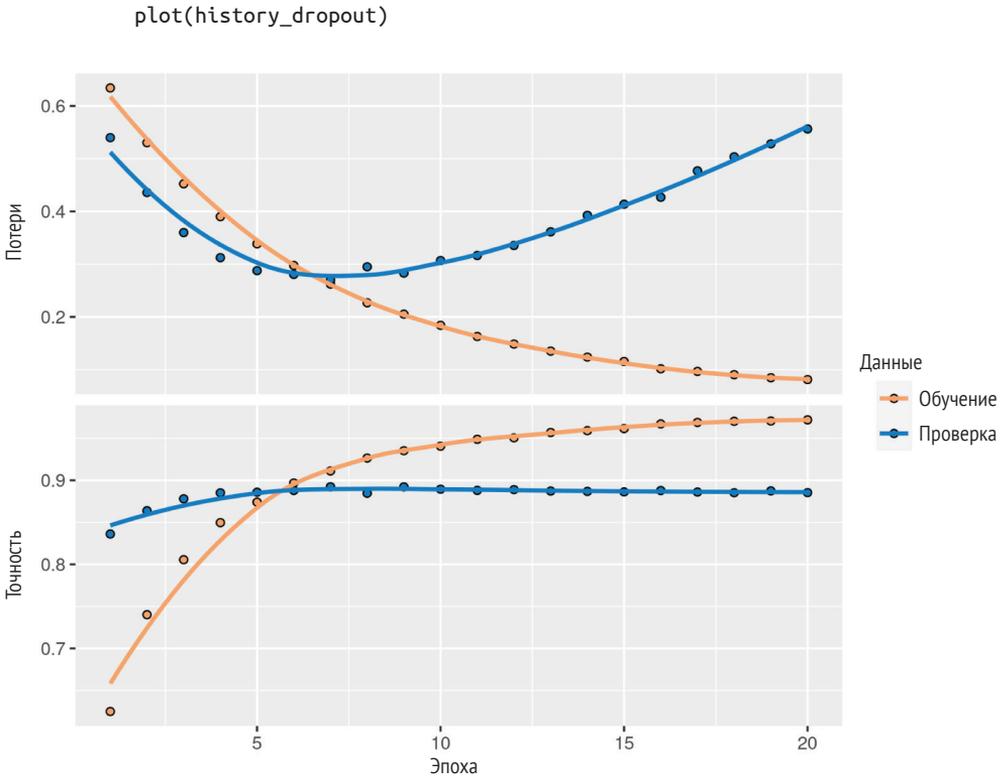
В Keras вы можете ввести прореживание в модель через слой `layer_dropout`, который применяет прореживание к выходным данным предшествующего слоя. Давайте добавим два слоя `layer_dropout` в модель IMDB, чтобы увидеть, насколько хорошо они справляются с уменьшением переобучения.

#### Листинг 5.16 Добавление отсева в модель IMDB

```
model <- keras_model_sequential() %>%
  layer_dense(16, activation = "relu") %>%
  layer_dropout(0.5) %>%
  layer_dense(16, activation = "relu") %>%
  layer_dropout(0.5) %>%
  layer_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 показан график результатов. Это явное улучшение по сравнению с исходной моделью – похоже, прореживание работает намного лучше, чем регуляризация L2, потому что показатель минимальной потери на проверочных данных улучшился.

#### Листинг 5.17 Создание графика для демонстрации влияния отсева на потерю проверки

```
plot(NULL,
      main = "Effect of Dropout on Validation Loss",
      xlab = "Epochs", xlim = c(1, history_original$params$epochs),
      ylab = "Validation Loss",
      ylim = range(c(history_original$metrics$val_loss,
                    history_dropout $metrics$val_loss)),
      panel.first = abline(v = 1:history_original$params$epochs,
                          lty = "dotted", col = "lightgrey"))
lines(history_original$metrics$val_loss, lty = 2)
lines(history_dropout $metrics$val_loss, lty = 1)
legend("topleft", lty = 1:2,
      legend = c("Потери модели с регуляризацией и отсевом",
                 "Потери исходной модели"))
```

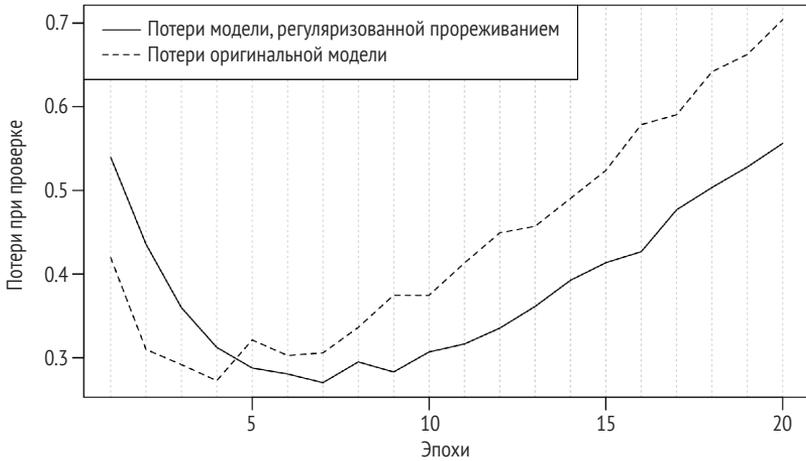


Рис. 5.21 Влияние прореживания на потерю при проверке

Завершая эту главу, вспомним наиболее распространенные способы достижения максимального обобщения и предотвращения переобучения в нейронных сетях:

- соберите больше обучающих данных или более качественные обучающие данные;
- сконструируйте лучшие признаки;
- уменьшите емкость модели;
- добавьте регуляризацию весов (для небольших моделей);
- добавьте прореживание.

## Краткие итоги главы

- Целью модели машинного обучения является *обобщение*: точное предсказание выхода для незнакомых входных данных. Это сложнее, чем кажется.
- Глубокая нейронная сеть достигает обобщения, подгоняя к данным параметрическую модель, которая может успешно *интерполировать* между обучающими выборками, – можно сказать, что такая модель изучила «скрытое многообразие» обучающих данных. Вот почему модели глубокого обучения могут работать только с входными данными, которые очень близки по форме и структуре к обучающим данным.
- Фундаментальной проблемой машинного обучения является *противоречие между оптимизацией и обобщением*: чтобы добиться обобщения, вы должны сначала добиться хорошей подгонки к обучающим данным, но улучшение соответствия вашей моде-

ли обучающим данным через некоторое время неизбежно начнет мешать обобщению. Каждая передовая методика глубокого обучения связана с нахождением компромисса между оптимизацией и обобщением.

- Способность моделей глубокого обучения к обобщению обусловлена тем, что им удастся аппроксимировать *скрытое многообразие* своих данных путем обучения и, как следствие, понимать новые входные данные посредством интерполяции.
- Очень важно иметь возможность точно оценить обобщающую способность вашей модели во время ее разработки. В вашем распоряжении целый ряд методов оценки, от простой проверки с отложенной выборкой до  $K$ -кратной перекрестной проверки и повторной  $K$ -кратной перекрестной проверки с перетасовкой. Не забывайте всегда хранить совершенно изолированный и неприкасаемый контрольный набор данных для окончательной оценки модели, потому что при многократном прогоне циклов обучение–проверка может произойти утечка информации из проверочных данных в модель.
- Когда вы начинаете работать над моделью, ваша цель состоит в том, чтобы в первую очередь получить модель, обладающую некоторой способностью к обобщению и способную к переобучению. В число лучших методик входят: настройка скорости обучения и размера пакета, использование специально подобранных архитектур, увеличение емкости модели или просто более длительное обучение.
- Как только модель начинает переобучаться, ваше внимание должно переключиться на улучшение обобщения за счет *регуляризации модели*. Вы можете уменьшить емкость своей модели, добавить прореживание слоев или регуляризацию весов, а также использовать раннюю остановку. Разумеется, наилучшим способом достижения высокой обобщающей способности модели остается использование большого и качественного обучающего набора.

# Обобщенный рабочий процесс машинного обучения

---

## *Эта глава охватывает следующие темы:*

- постановка задачи машинного обучения;
- этапы разработки прикладной модели;
- шаги по развертыванию вашей модели в производственной среде и ее поддержка.

В наших предыдущих примерах предполагалось, что у нас уже есть размеченный набор данных, с которого можно начать, и что мы можем немедленно приступить к обучению модели. В реальном мире такое случается редко. Вы не начинаете с набора данных; вы начинаете с задачи.

Представьте, что вы открываете стартап по машинному обучению и готовы братья за любые заказы. Вы зарегистрировали фирму, создали модный веб-сайт и сделали рассылку по всем известным вам потенциальным клиентам. Дела идут неплохо – вы сразу получили заказы на создание моделей глубокого обучения, которые должны решать следующие задачи:

- персонализированный поиск фотографий для социальной сети по обмену фотографиями – например, пользователь вводит слово «свадьба», и система находит все фотографии, сделанные им на свадьбах, без необходимости пометить снимки вручную;

- выявление спама и оскорбительного текстового содержания среди сообщений перспективного приложения для чата;
- построение системы музыкальных рекомендаций для пользователей онлайн-радиостанции;
- обнаружение попыток мошенничества с кредитными картами для сайта интернет-магазина;
- прогнозирование отклика на рекламу для принятия решения о том, какую рекламу показывать данному пользователю в данное время;
- выявление бракованного печенья на конвейерной ленте кондитерской фабрики;
- анализ спутниковых изображений для прогнозирования местоположения еще не найденных археологических памятников.

### Этические соображения

Иногда вам могут предлагать участие в сомнительных с этической точки зрения проектах, таких как «создание ИИ, который оценивает благонадежность человека по изображению его лица». Во-первых, сомнительна достоверность проекта: не понятно, откуда взялась уверенность, что благонадежность должна отражаться на чьем-то лице. Во-вторых, такая задача открывает двери для всевозможных этических проблем. Аннотирование набора данных для этой задачи будет означать лишь запись предубеждений людей, которые присваивают метки изображениям. Модели, которые вы обучите на таких данных, просто закодируют предвзятость людей в алгоритм черного ящика, который придаст чьим-то личным убеждениям видимость объективности. В таком технически безграмотном обществе, как наше, фраза «алгоритм ИИ сказал, что этому человеку нельзя доверять» странным образом кажется более весомой и объективной, чем «Джон Смит сказал, что этому человеку нельзя доверять», несмотря на то что первая является наукообразной формой последней. Такая модель будет отмыкать и продвигать в широкие массы худшие аспекты субъективного человеческого суждения с негативными последствиями для жизни реальных людей.

*Технологии никогда не бывают нейтральными.* Если ваша работа оказывает какое-то влияние на мир, это влияние имеет моральную составляющую: технические решения – это также и этические решения. Всегда думайте о ценностях, которых вы хотите придерживаться в своей работе.

Было замечательно, если бы вы могли получить доступ к хорошему набору данных с помощью метода `keras::dataset_mydataset()` и начать обучение различных моделей. К сожалению, в реальном мире вам придется начинать с нуля.

В этой главе вы ознакомитесь с универсальным планом действий, который можно использовать для решения любой задачи машинного обучения, подобной тем, что были в списке выше. Этот план объ-

единит и закрепит знания, полученные в главах 4 и 5, а также подготовит основу для знаний, которые вы получите в следующих главах.

Обобщенный рабочий процесс машинного обучения в целом состоит из трех частей.

- 1 *Постановка задачи.* Изучите область деятельности и бизнес-логику, лежащую в основе потребностей клиента. Соберите набор данных, поймите, что представляют собой данные, и выберите, по какому критерию вы будете измерять успех модели для этой задачи.
- 2 *Разработка модели.* Подготовьте имеющиеся данные, чтобы их можно было обработать с помощью модели машинного обучения, выберите протокол оценки модели и базовый уровень, который нужно превзойти, обучите первую модель, обладающую обобщающей способностью и способную к переобучению, а затем регуляризируйте и настраивайте свою модель, пока не достигнете наилучшего обобщения.
- 3 *Развертывание модели.* Представьте свою работу заказчику; отправьте модель на веб-сервер, мобильное приложение, веб-страницу или встроенное устройство; следите за поведением модели в реальных условиях; наконец, начните собирать данные, необходимые для построения модели следующего поколения.

Давайте рассмотрим эти шаги рабочего процесса более подробно.

## 6.1 *Постановка задачи*

Вы не сможете хорошо работать без глубокого понимания контекста того, что вы делаете. Почему ваш клиент пытается решить именно эту задачу? Какую ценность он получит от решения – как клиент будет использовать вашу модель и как она впишется его бизнес-процессы? Какие данные имеются или могут быть собраны? Какая типовая задача машинного обучения больше всего похожа на потребности клиента?

### 6.1.1 *Уточнение задачи*

Постановка задачи машинного обучения обычно включает в себя множество детальных обсуждений с заинтересованными сторонами. Вот вопросы, которые вы должны постоянно задавать себе:

Какой вид будут иметь входные данные? Что требуется предсказать? Вы сможете обучить сеть предсказывать что-либо только при наличии тренировочных данных: например, обучить сеть определять эмоциональную окраску отзывов к фильмам можно, если имеются отзывы и соответствующие аннотации. То есть доступность данных на данном этапе является ограничивающим фактором. Во

многих случаях вам придется прибегнуть к самостоятельному сбору и аннотированию новых наборов данных (о чем мы расскажем в следующем разделе).

К какому типу относится задача, стоящая перед вами? Бинарная классификация? Многоклассовая классификация? Скалярная регрессия? Векторная регрессия? Многозначная или многоуровневая классификация? Сегментация изображения? Рейтинг? Что-то иное, например кластеризация, генерация или обучение с подкреплением? В некоторых случаях машинное обучение является не самым лучшим способом осмысления данных, и вам следует использовать что-то другое, например традиционный статистический анализ:

- проект поискового движка для фотографий представляет собой задачу многоклассовой классификации с несколькими метками;
- проект обнаружения спама представляет собой задачу бинарной классификации. Если вы установите «оскорбительный контент» в качестве отдельного класса, это будет задача классификации с тремя классами;
- механизм музыкальных рекомендаций лучше строить не с помощью глубокого обучения, а с помощью факторизации матриц (коллаборативной фильтрации);
- проект по обнаружению мошенничества с кредитными картами представляет собой задачу бинарной классификации;
- проект прогнозирования отклика на рекламу представляет собой задачу скалярной регрессии;
- обнаружение бракованного печенья – это задача бинарной классификации, но она также нуждается в модели обнаружения объектов в качестве первого этапа, чтобы правильно выделить печенья в необработанных изображениях. Учтите, что набор методов машинного обучения, известный как «обнаружение аномалий», не подходит для этой ситуации!
- проект по поиску новых археологических памятников по спутниковым снимкам представляет собой задачу ранжирования сходства изображений: вам нужно найти новые изображения, которые больше всего похожи на известные археологические памятники.

Как выглядят существующие решения? Возможно, у вашего клиента уже есть созданный вручную алгоритм, который фильтрует спам или обнаруживает мошенничество с кредитными картами при помощи огромного количества вложенных операторов `if`. Возможно, в настоящее время человек отвечает за ручное управление рассматриваемым процессом – наблюдение за конвейерной лентой на кондитерской фабрике и ручное удаление бракованного печенья или создание плейлистов с рекомендациями песен, которые будут отправлены пользователям, предпочитающим конкретного исполнителя. Вы должны тщательно разобраться, какие системы уже существуют и как они работают.

Есть ли особые ограничения, с которыми вам придется иметь дело? Например, вы можете обнаружить, что трафик приложения, для которого вы создаете систему обнаружения спама, строго защищен сквозным шифрованием, так что модель обнаружения спама будет работать на телефоне конечного пользователя и должна быть обучена на внешнем наборе данных. Возможно, модель обнаружения бракованного печенья придется запускать на локальном заводском компьютере, а не на удаленном сервере. Вы должны понимать весь контекст, в который должна вписаться ваша работа.

В результате проведенного исследования вы должны знать, какими будут ваши входные данные, каковы будут ваши цели и к какому типу моделей машинного обучения относится задача. Помните о гипотезах, которые вы выдвигаете на этом этапе:

- гипотеза о том, что выходные данные можно предсказать по входным данным;
- гипотеза о том, что доступные данные (или которые вы вскоре соберете) достаточно информативны для изучения отношений между входными и выходными данными.

Пока у вас нет работающей модели, это всего лишь гипотезы, ожидающие подтверждения или опровержения. Не все задачи можно решить с помощью машинного обучения; 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 *Получение исходных данных*

Как только вы поймете характер задачи и узнаете, какими будут ваши входные данные и цели, пришло время для сбора данных – самой сложной, трудоемкой и дорогостоящей части большинства проектов машинного обучения:

- для системы поиска фотографий нужно, чтобы вы сначала составили набор меток, которые вы хотите классифицировать, – вы выбираете 10 000 общих категорий изображений. Затем вам нужно вручную аннотировать метками из этого набора сотни тысяч имеющихся изображений, загруженных пользователями;
- в проекте по обнаружению спама вы не можете использовать для обучения модели содержимое пользовательских чатов, поскольку оно полностью зашифровано. Вам необходимо получить доступ к отдельному набору данных из десятков тысяч нефильТРованных сообщений в социальных сетях и вручную пометить их как спам, оскорбление или приемлемый контент;

- для обучения механизма музыкальных рекомендаций вы можете просто использовать «лайки» ваших пользователей. Нет необходимости собирать новые данные. То же самое и с проектом прогнозирования отклика на рекламу: у вас есть обширный список рейтингов для ваших прошлых объявлений за несколько лет;
- для обучения модели, выявляющей бракованное печенье, вам придется установить камеры над конвейерными лентами и собрать десятки тысяч изображений, а затем кто-то должен будет вручную аннотировать эти изображения. На кондитерской фабрике работают люди, которые хорошо разбираются в печенье, и вам не составит труда обучить их аннотировать изображения.
- для проекта обработки спутниковых снимков потребуется группа археологов, чтобы собрать базу данных существующих объектов, представляющих интерес, и для каждого объекта вам нужно будет найти существующие спутниковые снимки, сделанные в различных погодных условиях. Чтобы получить хорошую модель, вам понадобятся тысячи разных сайтов.

В главе 5 вы узнали, что способность модели обобщать почти полностью зависит от свойств данных, на которых она обучается: количество точек данных, которые у вас есть, надежность ваших меток, качество ваших признаков. Хороший набор данных – это актив, заслуживающий труда и инвестиций. Если у вас есть дополнительные 50 часов, которые можно потратить на проект, наверняка имеет смысл собрать больше данных, а не искать дополнительные улучшения модели.

Тезис о том, что данные важнее алгоритмов, наиболее ярко выражен в статье 2009 года исследователей Google под названием «Непостижимая эффективность данных» (название является ссылкой к известной книге Юджина Вигнера «Непостижимая эффективность математики в естественных науках», опубликованной в 1960 г.). Это было сказано до того, как глубокое обучение стало популярным, но, что примечательно, рост глубокого обучения только повысил важность данных.

Если вы проводите обучение с учителем, то после того как вы соберете входные данные (например, изображения), вам понадобятся аннотации к ним (например, теги для этих изображений) – это цели, которые должна научиться прогнозировать ваша модель. Иногда аннотации могут извлекаться автоматически, например для задачи рекомендации музыки или задачи прогнозирования отклика на рекламу. Но часто приходится аннотировать свои данные вручную. Это трудоемкий процесс.

## **ИНВЕСТИЦИИ В ИНФРАСТРУКТУРУ АННОТИРОВАНИЯ ДАННЫХ**

Ваш процесс аннотирования данных будет определять качество целей, которые, в свою очередь, определяют качество вашей модели.

Тщательно продумайте ответы на следующие вопросы:

- стоит ли аннотировать данные самостоятельно?
- стоит ли использовать для аннотирования краудсорсинговую платформу, такую как Mechanical Turk?
- стоит ли пользоваться услугами специализированной компании по аннотированию данных?

Аутсорсинг потенциально может сэкономить вам время и деньги, но он лишает вас контроля. Использование платформы наподобие Mechanical Turk, вероятно, будет недорогим и хорошо масштабируемым решением, но аннотации могут оказаться довольно шумными.

Чтобы выбрать лучший вариант, уточните следующие особенности проекта, над которым вы работаете:

- должны ли аннотаторы данных быть экспертами в предметной области, или аннотировать данные может любой желающий? Метки для задачи классификации изображений кошек и собак могут быть выбраны кем угодно, но для задач классификации пород собак требуются специальные знания. Больше того, чтобы аннотировать рентгеновские снимки переломов костей, требуется профильное медицинское образование;
- если для аннотирования данных требуются специальные знания, можете ли вы обучить этому людей со стороны? Если нет, то где вы планируете найти профильных экспертов?
- имеете ли вы хотя бы общее представление о том, как специалисты назначают аннотации? Если это не так, вам придется обращаться с набором данных как с черным ящиком, и вы не сможете выполнять конструирование признаков вручную – это не критично, но может помешать.

Если вы решите аннотировать свои данные самостоятельно, спросите себя, какое программное обеспечение вы будете использовать для записи аннотаций. Возможно, вам придется разработать это программное обеспечение самостоятельно. Эффективное программное обеспечение для аннотирования данных экономит вам много времени, поэтому стоит уделить ему больше внимания в начале проекта.

## ОСТЕРЕГАЙТЕСЬ НЕРЕПРЕЗЕНТАТИВНЫХ ДАННЫХ

Чтобы модель машинного обучения понимала новые данные, они должны быть максимально похожи на данные из обучающего набора. Таким образом, очень важно, чтобы данные, используемые для обучения, были *репрезентативными* для производственных данных. Стремление к этому должно лечь в основу всей вашей работы по сбору данных.

Предположим, вы разрабатываете приложение, в котором пользователи могут сфотографировать тарелку с едой, чтобы узнать название блюда. Вы обучаете модель, используя фотографии из популяр-

ной среди гурманов социальной сети для обмена изображениями. Затем вы развертываете модель на производстве и получаете шквал негативных отзывов от разгневанных пользователей: ваше приложение дает неправильный ответ в 8 случаях из 10. В чем дело? Ведь точность модели на контрольном наборе была более 90%! Беглый взгляд на загруженные пользователями данные показывает, что снимки случайных блюд из случайных ресторанов, сделанных случайными смартфонами, совсем не похожи на хорошо освещенные, аппетитные изображения профессионального качества, на которых вы обучали модель: *ваши обучающие данные не были репрезентативными для производственных данных*. Это смертный грех – добро пожаловать в ад машинного обучения.

Если возможно, извлекайте данные непосредственно из среды, в которой будет использоваться ваша модель. Для модели классификации настроений на основе отзывов о фильмах следует искать данные в новых обзорах на IMDb, а не в обзорах ресторанов Yelp или обновлениях статуса в Twitter. Если вы хотите оценить эмоциональную окраску твита, начните со сбора и аннотирования реальных твитов от тех же пользователей, которых вы ожидаете в рабочей среде. Если обучение на производственных данных невозможно, добейтесь полного понимания того, чем отличаются ваши данные для обучения и производственные данные, и активно корректируйте эти различия.

Связанное с этим явление, о котором вам следует знать, – это *смена концепции*. Вы столкнетесь со сменой концепции почти во всех реальных задачах, особенно в тех, которые связаны с данными, созданными пользователями. Смена концепции возникает, когда свойства производственных данных изменяются со временем, что приводит к постепенному снижению точности модели. Механизм музыкальных рекомендаций, обученный в 2013 году, наверняка будет не очень эффективным сегодня. Точно так же набор данных IMDb, с которым вы уже знакомы, был собран в 2011 году, и модель, обученная на нем, скорее всего, будет хуже работать с обзорами 2020 года, потому что словарный запас, выражения и жанры фильмов со временем меняются. Смена концепции особенно остро проявляется в связательных сценариях, таких как обнаружение мошенничества с кредитными картами, где схемы мошенничества меняются практически каждый день. Чтобы справиться с быстрым изменением концепции, требуется непрерывный сбор данных, аннотирование и переобучение модели.

Не забывайте, что машинное обучение можно использовать только для запоминания шаблонов, которые присутствуют в ваших обучающих данных. Вы можете узнать только то, что видели раньше. Использование машинного обучения, обученного на прошлых данных, для прогнозирования будущего предполагает, что будущее будет вести себя как прошлое. Часто это не так.

### Проблема ошибки отбора

Особенно коварным и распространенным случаем нерепрезентативных данных является *систематическая ошибка выборки* (ошибка отбора). Систематическая ошибка возникает, когда ваш процесс сбора данных взаимодействует с тем, что вы пытаетесь предсказать, что приводит к искажению выборки. Широко известен исторический пример президентских выборов в США в 1948 году. В ночь перед выборами газета «Чикаго Трибьюн» напечатала заголовок «ДЬЮИ ПОБЕЖДАЕТ ТРУМЭНА». На следующее утро Трумэн вышел победителем. Редактор газеты слишком доверился результатам телефонного опроса, но владельцы домашних телефонов в 1948 году не были случайной репрезентативной выборкой голосующего населения. В целом они представляли богатую консервативную прослойку общества и голосовали за Дьюи, кандидата от республиканцев.



«ДЬЮИ ПОБЕЖДАЕТ ТРУМЭНА»: известный пример ошибочной выборки

В настоящее время в каждом телефонном опросе учитывается систематическая ошибка выборки. Это не означает, что ошибка отбора ушла в прошлое в политических опросах, – это далеко не так. Но, в отличие от 1948 года, социологи знают об этом и предпринимают шаги, чтобы исправить ситуацию.

### 6.1.3 Добейтесь понимания данных

С вашей стороны будет большой ошибкой обращаться с набором данных как с черным ящиком. Прежде чем приступить к обучению моделей, вы должны изучить и визуализировать свои данные, чтобы получить представление о том, что делает их предсказательными, что поможет сконструировать признаки и выявить потенциальные проблемы:

- если ваши данные содержат изображения или текст на естественном языке, просмотрите несколько образцов (и их метки) напрямую;
- если ваши данные содержат числовые признаки, рекомендуется построить гистограмму значений признаков, чтобы получить представление о рабочем диапазоне и частотности различных значений;
- если ваши данные включают информацию о местоположении, нанесите ее на карту. Наблюдаются ли какие-либо четкие закономерности?
- в отдельных образцах отсутствуют значения некоторых признаков? Если это так, вам придется устранить замеченные недостатки при подготовке данных (об этом будет рассказано в следующем разделе);
- если ваша задача связана с классификацией, выведите количество экземпляров каждого класса в ваших данных. Представлены ли классы примерно поровну? Если нет, вам нужно будет учитывать этот дисбаланс;
- проверьте данные на возможность *просачивания цели* (target leaking): наличие в ваших данных признаков, которые содержат априорную информацию о целях и которые могут быть недоступны в рабочей среде. Если вы обучаете модель предсказания рака на медицинских картах пациентов и эти карты содержат запись «диагностирован рак», то ваши цели искусственно просачиваются в данные (запись о диагнозе служит «подсказкой» для модели). Всегда спрашивайте себя, каждый ли признак в ваших данных будет доступен в той же форме в производственной среде.

#### 6.1.4 *Выберите меру успеха*

Чтобы что-то контролировать, нужно уметь за этим наблюдать. Чтобы добиться успеха в проекте, вы должны сначала определить, что вы подразумеваете под качеством прогноза модели. Точность? Повторяемость и полноту? Уровень удержания клиентов? Ваша метрика успеха будет определять все технические решения, которые вы принимаете на протяжении работы над проектом. Она должна напрямую соответствовать вашим целям более высокого уровня, таким как показатели вашего клиента в бизнесе.

Для задач сбалансированной классификации, где каждый класс равновероятен, обычными показателями являются точность и *площадь под кривой рабочей характеристики приемника* (area under receiver operating characteristic curve, сокращенно ROC AUC). Для задач с дисбалансом классов, задач ранжирования или классификации с несколькими метками вы можете использовать точность и полноту, а также взвешенную форму точности или ROC AUC. Нередко приходится вырабатывать собственную метрику для измерения успеха.

Чтобы получить представление о разнообразии показателей успеха машинного обучения и о том, как они соотносятся с различными предметными областями, полезно просмотреть конкурсы по науке о данных на Kaggle (<https://kaggle.com>); они демонстрируют широкий спектр проблем и показателей оценки.

## 6.2 Разработка модели

Выбрав метрику успеха, вы сможете приступить к разработке модели. В большинстве учебных пособий и исследовательских проектов предполагается, что это единственный шаг – пропущены этапы постановки задачи и сбора данных, которые, как предполагается, уже выполнены (интересно, кем?), а также пропущено развертывание и обслуживание модели, которыми, как предполагается, снова занимается кто-то другой. На самом деле разработка модели – это лишь один из этапов рабочего процесса машинного обучения, и, если вы спросите меня, он не самый сложный. Самое сложное в машинном обучении – постановка задачи, сбор, аннотирование и очистка данных. Так что выше нос – то, что будет дальше, намного проще по сравнению с этим!

### 6.2.1 Подготовка данных

Как вы помните, модели глубокого обучения обычно не принимают необработанные данные. Предварительная обработка направлена на то, чтобы сделать имеющиеся исходные данные более доступными для нейронных сетей. Сюда входят векторизация, нормализация или обработка пропущенных значений. Многие методы предварительной обработки зависят от предметной области (например, к текстовым данным или изображениям могут применяться особые приемы обработки); мы рассмотрим их в следующих главах, когда займемся практическими примерами. А сейчас рассмотрим базовые методы, общие для всех предметных областей.

#### ВЕКТОРИЗАЦИЯ

Все входные данные и цели в нейронной сети обычно должны быть представлены в форме тензоров с плавающей запятой (или, в некоторых случаях, тензоров целых чисел или строк). Какие бы данные вы ни намеревались обработать – звук, изображения или текст, – вы должны сначала преобразовать их в тензоры. Этот шаг называется *векторизацией данных*. Например, в двух примерах классификации текста в главе 4 мы начали с текста, представленного в виде списков целых чисел (обозначающих последовательности слов), и использовали унитарное кодирование, чтобы преобразовать их в тензор данных `float32`. В примерах классификации цифр и прогнозирования

цен на жилье данные поступали в векторизованном виде, поэтому мы смогли пропустить этот шаг.

## НОРМАЛИЗАЦИЯ ЗНАЧЕНИЙ

В примере с классификацией цифр MNIST из главы 2 мы начали с изображений, пиксели которых закодированы как целые числа в диапазоне 0–255 в соответствии со значениями оттенков серого. Прежде чем передать эти данные в нашу сеть, мы разделили их на 255, чтобы получить значения с плавающей запятой в диапазоне 0–1. Точно так же при прогнозировании цен на жилье мы начали с признаков, которые занимали различные диапазоны: одни признаки имели небольшие значения с плавающей запятой, а другие – довольно большие целочисленные значения. Прежде чем загрузить эти данные в нашу сеть, нам пришлось нормализовать по отдельности каждый признак, чтобы он имел стандартное отклонение, равное 1, и среднее значение, равное 0.

В целом небезопасно вводить в нейронную сеть данные, которые принимают относительно большие значения (например, многочисленные целые числа, которые намного больше, чем начальные значения, принимаемые весами сети), или данные, которые являются разнородными (например, данные, где один признак находится в диапазоне 0–1, а другой – в диапазоне 100–200). Это может вызвать большие обновления градиента, которые помешают сходимости сети. Правильные обучающие данные должны иметь следующие характеристики:

- *маленькие значения* – как правило, большинство значений должны находиться в диапазоне 0–1;
- *однородность* – все признаки должны принимать значения примерно в одном диапазоне.

Кроме того, широко распространена и полезна следующая более строгая практика нормализации, хотя она не всегда необходима (например, мы не делали этого в примере с классификацией цифр):

- независимо нормализовать каждый признак, чтобы он имел среднее значение 0;
- независимо нормализовать каждый признак, чтобы стандартное отклонение было равно 1.

Это легко сделать с помощью функции `scale()`:

```
x <- scale(x)
```

Предполагаем, что `x` представляет собой двумерную матрицу данных формы (samples, features)

## ОБРАБОТКА ОТСУТСТВУЮЩИХ ЗНАЧЕНИЙ

Иногда в ваших данных могут отсутствовать значения. Например, в примере с ценой на жилье первым признаком был уровень преступности на душу населения. Что, если этот признак указан не для

всех образцов? Тогда вы столкнетесь с отсутствующими значениями в обучающих или тестовых данных. Вы можете полностью отказаться от такого признака, но вам не обязательно это делать:

- если признак является категориальным, можно безопасно создать новую категорию «значение отсутствует». Модель автоматически узнает, что это означает в отношении целей;
- если признак является числовым, избегайте ввода произвольного значения, такого как  $\theta$ , потому что это может создать разрыв в скрытом пространстве, образованном вашими признаками, что затруднит обобщение модели, обученной на таких данных. Вместо этого можно заменить отсутствующее значение средним или медианным значением объекта в наборе данных. Вы также можете научить модель прогнозировать значение признака с учетом значений других признаков.

Обратите внимание: если вы допускаете отсутствие категориальных признаков в контрольных данных, но сеть была обучена на данных без каких-либо пропусков, она не научится игнорировать пропущенные значения! В этой ситуации следует искусственно сгенерировать обучающие выборки с отсутствующими элементами: несколько раз скопировать произвольные обучающие выборки и отбросить некоторые категориальные признаки, которые, как вы ожидаете, могут отсутствовать в контрольных данных.

## 6.2.2 Выбор протокола оценки

Как вы узнали из предыдущей главы, конечной целью процесса обучения модели является достижение обобщения, и каждое решение, которое вы будете принимать в процессе разработки модели, будет основываться на *метриках проверки*, оценивающих эффективность обобщения. Назначение вашего протокола проверки состоит в том, чтобы точно определить, какой будет выбранная вами метрика успеха (например, точность) на основе фактических производственных данных. Надежность этого процесса имеет решающее значение для построения полезной модели. В главе 5 мы рассмотрели три общих протокола оценки:

- *отложенный проверочный набор* – лучшее решение, когда у вас много данных;
- *K-кратная перекрестная проверка* – это правильный выбор, когда у вас недостаточно данных для формирования надежного проверочного набора;
- *повторная K-кратная перекрестная проверка* – необходима для выполнения точной оценки модели, когда доступно очень мало данных.

Выберите один из этих протоколов. В большинстве случаев достаточно хорошо работает первый способ. Однако не забывайте о ре-

*презентативности* вашего проверочного набора и следите за тем, чтобы между обучающим и проверочным наборами не было пересекающихся выборок.

### 6.2.3 *Как превзойти простой базовый уровень*

Когда вы начнете работать непосредственно над моделью, ваша первоначальная цель – достичь статистической мощности, то есть разработать небольшую модель, способную превзойти простой базовый уровень. На этом этапе вы должны сосредоточиться на трех самых важных вещах:

- *конструирование признаков* – отфильтруйте неинформативные признаки (*отбор признаков*) и используйте свои знания о задаче для разработки новых признаков, которые могут оказаться полезными;
- *выбор правильной априорно обоснованной архитектуры* – какой тип архитектуры модели вы будете использовать? Полносвязная, сверточная, рекуррентная или трансформер? Стоит ли вообще использовать глубокое обучение или сначала лучше попробовать что-то другое?
- *выбор подходящей конфигурации обучения* – какую функцию потерь следует использовать? Каковы размер пакета и скорость обучения?

Для большинства задач вы можете начать с существующих решений. Вы далеко не первый, кто пытается создать детектор спама, механизм музыкальных рекомендаций или классификатор изображений. Обязательно изучите существующие методы конструирования признаков и архитектуры моделей, которые, скорее всего, достаточно хорошо решают вашу задачу.

Стоит отметить, что не всегда возможно достичь желаемого обобщения. Если вы не можете превзойти простой базовый уровень, после того как попробовали несколько разумных архитектур, возможно, ответ на вопрос, который вы задаете, отсутствует во входных данных. Начиная работу над любой моделью глубокого обучения, вы выдвигаете две гипотезы:

- вы предполагаете, что выход может быть предсказан на основе входных данных;
- вы предполагаете, что имеющиеся данные достаточно информативны, чтобы понять взаимосвязь между входами и выходами.

Вполне может быть, что эти гипотезы ложны, и в этом случае вы должны вернуться к самому началу и найти либо другие данные, либо другой способ решения задачи.

### Выбор правильной функции потерь

Часто бывает невозможно напрямую оптимизировать метрику, которая измеряет успех модели в решении задачи. Иногда нет простого способа превратить метрику в функцию потерь; функции потерь, в конце концов, должны быть вычислимы для мини-пакета данных (в идеале функция потерь должна быть вычислима всего лишь для одной точки данных) и должны быть дифференцируемы (в противном случае вы не сможете использовать обратное распространение для обучения своей сети). Например, популярная метрика классификации ROC AUC не может быть оптимизирована напрямую. Поэтому в задачах классификации обычно оптимизируют прокси-метрику ROC AUC, такую как кросс-энтропия. Смысл в том, что чем ниже кросс-энтропия, тем выше будет ROC AUC.

Следующая таблица поможет вам выбрать функцию активации последнего слоя и функцию потерь для нескольких распространенных типов задач.

Выбор правильной функции активации и функции потерь последнего слоя

Тип задачи	Активация последнего слоя	Функция потерь
Бинарная классификация	sigmoid	binary_crossentropy
Многоклассовая классификация с одиночным аннотированием	softmax	categorical_crossentropy
Многоклассовая классификация с множественным аннотированием	sigmoid	binary_crossentropy

## 6.2.4 Масштабирование: разработка модели, способной к переобучению

Как только вы получили модель, обладающую статистической мощностью (т. е. способную выдавать прогнозы, надежно выходящие за рамки простой случайности), возникает вопрос: достаточно ли мощна ваша модель? Достаточно ли слоев и параметров для правильного моделирования задачи? Например, модель логистической регрессии обладает статистической мощностью в задаче MNIST, но ее недостаточно для правильного решения задачи. Помните, что главное противоречие в машинном обучении возникает между оптимизацией и обобщением. Идеальная модель – это та, которая стоит прямо на границе между недообучением и переобучением, между недостаточной и избыточной статистической мощностью. Чтобы понять, где проходит эта граница, нужно сначала ее пересечь.

Чтобы выяснить, насколько большая модель вам понадобится, вы должны разработать модель, которая гарантированно переобучается. Способы достижения этого довольно просты, как вы узнали из главы 5:

- 1 добавляем слои;
- 2 делаем слои больше;
- 3 обучаем модель на большем количестве эпох.

Всегда отслеживайте потери, а также значения любых метрик, которые вас интересуют, отдельно при обучении и при проверке. Как только вы видите, что точность модели на проверочных данных начинает ухудшаться, вы добились переобучения.

## 6.2.5 *Регуляризация и настройка модели*

Как только вы достигнете статистической мощности и переобучения, вы поймете, что находитесь на правильном пути. На этом этапе вашей целью становится максимизация обобщающей способности модели.

Эта фаза займет больше всего времени: вы будете неоднократно изменять свою модель, обучать ее, оценивать на проверочных данных (не на контрольных!), изменять ее снова и повторять обучение, пока модель не станет настолько хороша, насколько это возможно. Вот некоторые вещи, которые вы должны попробовать:

- разные архитектуры; добавление или удаление слоев;
- добавление прореживания;
- если ваша модель небольшая, добавьте регуляризацию L1 или L2;
- разные гиперпараметры (например, количество элементов на уровне или скорость обучения оптимизатора), чтобы найти оптимальную конфигурацию;
- при необходимости повторите обработку данных или конструирование признаков: соберите и аннотируйте больше данных, разработайте более подходящие признаки или удалите признаки, которые кажутся неинформативными.

Большую часть этой работы можно автоматизировать с помощью инструмента для *автоматической настройки гиперпараметров*, такого как KerasTuner. Мы рассмотрим его в главе 13.

Помните о следующем: каждый раз, когда вы используете обратную связь от процесса проверки для настройки своей модели, вы впускаете информацию о процессе проверки в модель. Ничего страшного, если вы прошли через цикл настройки несколько раз; но повторяемые многократно итерации настройки в конечном итоге приведут к тому, что ваша модель приспособится к процессу проверки (даже если модель не обучается напрямую на проверочных данных). Это делает процесс оценки менее надежным.

Добившись удовлетворительной конфигурации модели, вы можете обучить финальную производственную модель на всех доступных данных (обучающих и проверочных) и оценить ее в последний раз на контрольном наборе. Если выяснится, что точность модели на контрольном наборе значительно хуже, чем измеренная ранее точность

на проверочных данных, это может означать одно из двух: или ваш протокол оценивания не был надежным, или вы допустили подгонку модели к проверочным данным при настройке параметров. В этом случае будет полезно переключиться на более надежный протокол оценивания (например, повторную  $K$ -кратную проверку).

## 6.3 Развертывание модели

Итак, ваша модель успешно прошла окончательную оценку на контрольном наборе – теперь она готова к развертыванию и началу продуктивной жизни.

### 6.3.1 Представление модели заказчику

Успех и доверие клиентов держатся на постоянном удовлетворении или превышении ожиданий людей. Техническая часть системы, которую вы поставляете, – это только одна половина успеха; вторая половина заключается в формировании адекватных ожиданий заказчика перед запуском.

Ожидания неспециалистов в отношении систем ИИ часто нереалистичны. Например, они могут ожидать, что система «понимает» свою задачу и способна проявлять человеческий здравый смысл в контексте задачи. Чтобы исключить это заблуждение, вам следует позаботиться о том, чтобы показать несколько примеров неправильной работы вашей модели (например, показать, как выглядят неправильно классифицированные образцы, особенно те, для которых неправильная классификация кажется неожиданной).

Они также могут рассчитывать на точность на уровне человека, особенно для процессов, которые ранее выполнялись людьми. Большинство моделей машинного обучения из-за того, что они не идеально обучены аппроксимировать метки, созданные человеком, почти не достигают этого уровня. Вы должны четко сформулировать ожидания относительно точности модели. Избегайте использования абстрактных утверждений, таких как «модель имеет точность 98 %» (которые большинство людей мысленно округляют до 100 %), и старайтесь говорить, например, о ложноотрицательных и ложноположительных коэффициентах. Вы можете сказать: «С этими настройками модель обнаружения мошенничества будет давать 5 % ложноотрицательных результатов и 2,5 % ложноположительных результатов. Каждый день в среднем 200 легальных транзакций будут помечаться как мошеннические и отправляться на проверку вручную, а в среднем 14 мошеннических транзакций будут пропущены. В среднем будет правильно обнаружено 266 мошеннических транзакций». Четко свяжите показатели точности модели с бизнес-целями.

Вы также должны обсудить с заинтересованными сторонами выбор ключевых параметров срабатывания, например порог вероятности, при котором транзакция должна быть отклонена (разные пороги будут давать разные уровни ложноотрицательных и ложноположительных результатов). Такие решения предполагают компромиссы, которые могут быть реализованы только при глубоком понимании бизнес-контекста.

### 6.3.2 Передача модели заказчику

Проект машинного обучения не заканчивается, когда вы наконец получаете возможность сохранить обученную модель. Вы редко запускаете в производство точно такой же объект модели, которым вы манипулировали во время обучения. Во-первых, вам может понадобиться экспортировать свою модель на какую-то другую платформу, кроме R:

- ваша производственная среда может вообще не поддерживать R, например если это мобильное приложение или встроена система;
- если остальная часть приложения написана не на R (это может быть JavaScript, C++ и т. д.), использование R для обслуживания модели может привести к значительным накладным расходам.

Во-вторых, поскольку ваша производственная модель будет использоваться только для вывода прогнозов (этап, называемый *логическим выводом*), а не для обучения, у вас есть возможность для различных оптимизаций, которые могут сделать модель быстрее и уменьшить объем занимаемой памяти. Давайте кратко рассмотрим различные доступные варианты развертывания модели.

#### РАЗВЕРТЫВАНИЕ МОДЕЛИ КАК REST API

Это, пожалуй, самый распространенный способ превратить модель в продукт: установить TensorFlow на сервере или в облаке и запросить прогнозы модели через REST API. Вы можете создать собственное обслуживающее приложение, используя что-то вроде Shiny (или любую другую библиотеку R для веб-разработки) или пакет R `tfdeploy`, который использует собственную библиотеку TensorFlow для доставки моделей в виде API, называемую TensorFlow Serving (<https://www.tensorflow.org/tfx/guide/serving>). С помощью `tfdeploy` и TensorFlow Serving вы можете развернуть модель Keras за считанные минуты.

Вы должны использовать этот способ развертывания, когда:

- приложение, которое будет использовать прогноз модели, будет иметь надежный доступ к Интернету (что вполне очевидно). Например, если ваше приложение является мобильным, получение прогнозов через удаленный API означает, что приложе-

ние нельзя будет использовать в режиме полета или в местности с плохой мобильной связью;

- приложение не имеет строгих требований к задержке: отправка запроса, вывод и получение ответа обычно занимают около 500 мс;
- входные данные, отправленные для вывода, не являются критическими в плане приватности: данные должны быть доступны на сервере в расшифрованном виде, потому что они должны быть видны модели (но учтите, что вы должны использовать SSL-шифрование для HTTP-запроса и ответа).

Например, проекты системы поиска изображений, системы рекомендаций музыки, обнаружения мошенничества с кредитными картами и анализа спутниковых снимков хорошо подходят для обслуживания через REST API.

Важный вопрос при развертывании модели в качестве REST API заключается в том, хотите ли вы разместить код самостоятельно или использовать полностью управляемую стороннюю облачную службу. Например, продукт Google под названием Cloud AI Platform позволяет просто загрузить модель TensorFlow в облачное хранилище Google (GCS) и предоставляет конечную точку API для запроса к ней. Он сам позаботится о многих практических деталях, таких как пакетные прогнозы, балансировка нагрузки и масштабирование.

## РАЗВЕРТЫВАНИЕ МОДЕЛИ НА УСТРОЙСТВЕ

Иногда бывает нужно, чтобы ваша модель работала на том же устройстве, на котором запущено приложение, использующее ее, – это может быть смартфон, встроенный процессор ARM на роботе или микроконтроллер на крошечном устройстве. Наверняка вы видели фотокамеру, способную автоматически обнаруживать людей и лица в сценах, на которые вы ее навели: скорее всего, это была небольшая модель глубокого обучения, работающая непосредственно на камере.

Вы должны использовать этот способ развертывания, когда:

- ваша модель имеет строгие ограничения по задержке или должна работать в среде с медленной связью. Если вы создаете иммерсивное приложение дополненной реальности, задержки при обращении к удаленному серверу неприемлемы;
- ваша модель достаточно маленькая и может работать в условиях скромной памяти и вычислительной мощности целевого устройства. Вы можете использовать инструментарий оптимизации TensorFlow, чтобы облегчить перенос модели на устройство ([http://www.tensorflow.org/model\\_optimization](http://www.tensorflow.org/model_optimization));
- достижение максимально возможной точности не является критически важным требованием для вашей задачи. Всегда существует компромисс между эффективностью выполнения и точностью, поэтому ограничения по памяти и мощности часто требуют от вас поставки модели, которая не так хороша, как

лучшая модель, которую вы могли бы запустить на большом графическом процессоре;

- входные данные строго конфиденциальны и поэтому не должны быть расшифрованы на удаленном сервере.

Наша модель обнаружения спама должна работать на смартфоне конечного пользователя как часть приложения чата, поскольку сообщения полностью зашифрованы и не могут быть прочитаны удаленной моделью. Аналогично модель обнаружения бракованного печенья имеет строгие ограничения по задержке, и ее необходимо запускать на заводе. К счастью, в этом случае у нас нет ограничений по мощности или пространству, поэтому мы можем запустить модель на графическом процессоре.

Чтобы развернуть модель Keras на смартфоне или встроенном устройстве, вам подойдет решение TensorFlow Lite (<http://www.tensorflow.org/lite>). Это платформа для эффективной реализации глубокого обучения на маломощных устройствах, которая работает на смартфонах Android и iOS, а также на компьютерах на базе ARM64, Raspberry Pi или некоторых микроконтроллерах. Она включает в себя конвертер, который может напрямую преобразовать вашу модель Keras в формат TensorFlow Lite.

## РАЗВЕРТЫВАНИЕ МОДЕЛИ В БРАУЗЕРЕ

Глубокое обучение часто используется в приложениях JavaScript для браузеров, работающих на компьютере пользователя. Хотя обычно приложение может запрашивать удаленную модель через REST API, существуют определенные преимущества запуска модели непосредственно в браузере на компьютере пользователя (применяя ресурсы графического процессора, если они доступны).

Используйте этот способ развертывания, если:

- вы хотите применить вычислительные ресурсы конечного пользователя, что может значительно снизить затраты на сервер;
- вводимые данные должны оставаться на компьютере или телефоне конечного пользователя. Например, в нашем проекте по обнаружению спама веб-версия и локальная версия приложения чата (реализованного как кросс-платформенное приложение, написанное на JavaScript) должны использовать локально запускаемую модель;
- ваше приложение имеет строгие ограничения по задержке. Хотя модель, работающая на ноутбуке или смартфоне конечного пользователя, скорее всего, будет медленнее, чем модель, работающая на большом графическом процессоре на вашем собственном сервере, у вас нет лишних 100 мс на передачу данных по сети туда и обратно;
- вам нужно, чтобы ваше приложение продолжало работать без подключения к сети, после того как модель была загружена и кеширована.

Вы можете использовать этот вариант, только если ваша модель достаточно мала, чтобы не перегружать процессор, графический процессор или оперативную память ноутбука или смартфона пользователя. Кроме того, поскольку вся модель будет загружена на устройство пользователя, вы должны убедиться, что никакая информация о модели не является конфиденциальной. Помните о том, что при наличии обученной модели глубокого обучения обычно можно восстановить некоторую информацию об обучающих данных, поэтому лучше не раскрывать обученную модель, если она была обучена на конфиденциальных данных.

Для развертывания модели в JavaScript экосистема TensorFlow предлагает TensorFlow.js (<http://www.tensorflow.org/js>), библиотеку JavaScript для глубокого обучения, которая реализует почти все возможности API Keras (изначально разработанную под рабочим названием WebKeras), а также многие API-интерфейсы TensorFlow более низкого уровня. Вы можете легко импортировать сохраненную модель Keras в TensorFlow.js, чтобы запустить ее как часть вашего браузерного приложения JavaScript или настольного приложения Electron.

## ОПТИМИЗАЦИЯ МОДЕЛИ

Оптимизация вашей модели для вывода особенно важна при развертывании в среде со строгими ограничениями на доступную мощность и память (смартфоны и встроенные устройства) или для приложений с низкими требованиями к задержке. Вы всегда должны стремиться оптимизировать свою модель перед импортом в TensorFlow.js или экспортом в TensorFlow Lite.

Вы можете применить два популярных метода оптимизации:

- *отсечение весов* (weight pruning) – не все коэффициенты в тензоре весов вносят одинаковый вклад в прогнозы. Можно существенно уменьшить количество параметров в слоях вашей модели, оставив только самые значимые из них. Это уменьшает объем памяти и вычислений вашей модели при незначительном ухудшении показателей точности. Величина отсечения определяется компромиссом между размером и точностью;
- *квантование весов* (weight quantization) – модели глубокого обучения обучаются с использованием весов одинарной точности с плавающей запятой (float32). Однако можно преобразовать веса в 8-битные целые числа со знаком (int8), чтобы получить модель, предназначенную только для вывода, которая в четыре раза меньше по размеру, но остается близкой к точности исходной модели.

Экосистема TensorFlow включает набор инструментов для отсечения и квантования весов ([http://www.tensorflow.org/model\\_optimization](http://www.tensorflow.org/model_optimization)), глубоко интегрированный с API Keras.

### 6.3.3 *Мониторинг модели в рабочей среде*

Допустим, вы экспортировали модель логического вывода, интегрировали ее в свое приложение и выполнили пробный прогон на производственных данных – модель вела себя именно так, как вы ожидали. Вы написали модульные тесты, а также код ведения журнала и мониторинга состояния – отлично. Теперь пришло время нажать большую красную кнопку и выполнить развертывание в рабочей среде.

Но и это еще не все. После того как вы развернули модель, вам нужно продолжать отслеживать ее поведение, ее точность с новыми данными, взаимодействие с остальной частью приложения и возможное влияние на бизнес-показатели:

- увеличилась ли активность пользователей вашей онлайн-радиостанции после развертывания новой системы музыкальных рекомендаций? Увеличилась ли средняя доля переходов по рекламным объявлениям после перехода на новую модель прогнозирования отклика на рекламу? Вам стоит подумать об использовании рандомизированного A/B-тестирования, чтобы изолировать влияние самой модели от других изменений: рабочее подмножество случаев должно проходить через новую модель, тогда как контрольное подмножество должно придерживаться старого процесса. Как только будет обработано достаточно много случаев, разница в результатах между ними, вероятно, будет связана с моделью;
- если возможно, проводите регулярную ручную проверку прогнозов модели на производственных данных. Как правило, можно повторно использовать ту же инфраструктуру, что и для аннотирования данных: отправить некоторую часть производственных данных для аннотирования вручную и сравнить прогнозы модели с новыми аннотациями. Например, вы обязательно должны сделать это для поисковой системы изображений и системы отбраковки печенья;
- если ручной аудит невозможен, рассмотрите альтернативные способы оценки, такие как опросы пользователей (например, в случае системы пометки спама и оскорбительного контента).

### 6.3.4 *Поддержка и обновление модели*

Наконец, ни одна модель не вечна. Вы уже знаете о смене концепции: со временем характеристики производственных данных будут меняться, постепенно снижая точность и актуальность вашей модели. Срок службы вашей системы музыкальных рекомендаций будет исчисляться неделями. Для системы обнаружения мошенничества с кредитными картами это будут дни; система для поиска изображений в лучшем случае продержится пару лет.

Как только ваша модель будет развернута в производстве, вы должны подготовиться к обучению следующего поколения, которое заменит ее. Для этого нужно:

- следить за изменениями в производственных данных. Не появились ли новые признаки? Не пора ли расширить или иным образом изменить набор меток?
- продолжать собирать и аннотировать данные, а также постоянно улучшать процесс аннотирования. В частности, вам следует уделять особое внимание сбору образцов, которые кажутся сложными для вашей текущей модели, – такие образцы, скорее всего, помогут улучшить точность в будущем.

На этом универсальный рабочий процесс машинного обучения завершается – согласен, вам нужно помнить о многих вещах. Чтобы стать экспертом, требуется время и опыт, но не волнуйтесь: вы уже знаете и умеете намного больше, чем несколько глав назад. Теперь вы знакомы с общей картиной – со всем спектром проектов машинного обучения. Хотя большая часть этой книги посвящена разработке моделей, теперь вы знаете, что это лишь часть обширного рабочего процесса. Всегда держите перед глазами общую картину!

## Краткие итоги главы

- Приступая к новому проекту машинного обучения, сначала определите задачу:
  - изучите общий контекст, в котором будет работать модель, – какова конечная цель и каковы ограничения?
  - соберите и аннотируйте набор данных, добейтесь глубокого понимания всех аспектов данных;
  - выберите метрику успеха будущей модели – какие показатели вы будете отслеживать в своих проверочных данных?
- Сформулировав четкую задачу и подготовив соответствующий набор данных, приступайте к разработке модели:
  - обработайте данные;
  - выберите свой протокол оценки: отложенная проверочная выборка,  $K$ -кратная перекрестная проверка? Какую часть данных следует использовать для проверки?
  - добейтесь статистической мощности: модель должна превзойти тривиальный базовый уровень;
  - масштабирование: разработайте модель, которая способна к переобучению;
  - регуляризируйте модель и настройте ее гиперпараметры, ориентируясь на результаты оценки на проверочном наборе. Многие исследования в области машинного обучения, как правило, сосредоточены только на этом шаге, но помните об общей картине.

- Когда ваша модель готова и демонстрирует хорошую точность на контрольных данных, приступайте к развертыванию:
  - сформируйте у заказчика и других заинтересованных лиц адекватные ожидания;
  - оптимизируйте окончательную модель и отправьте ее в выбранную среду развертывания – веб-сервер, мобильное устройство, браузер, встроенное устройство и т. д.
  - отслеживайте поведение вашей модели в производственной среде и продолжайте собирать данные, чтобы вовремя разработать модель следующего поколения.

# Работа с Keras: углубленные навыки

---

## **Эта глава охватывает следующие темы:**

- создание моделей Keras с помощью `keras_model_sequential()`, функционального API и подклассов моделей;
- использование встроенных циклов обучения и оценки Keras;
- применение обратных вызовов Keras для настройки обучения;
- использование TensorBoard для мониторинга показателей обучения и оценки;
- разработка циклов обучения и оценки с нуля.

У вас уже есть некоторый опыт работы с Keras – вы знакомы с последовательной моделью, полносвязными слоями и встроенными API для обучения, оценки и вывода – `compile()`, `fit()`, `evaluate()` и `predict()`. В главе 3 вы даже узнали, как использовать `new_layer_class()` для создания пользовательских слоев и как использовать предоставляемый TensorFlow метод `GradientTape()` для пошаговой реализации цикла обучения.

В следующих главах мы подробно рассмотрим компьютерное зрение, прогнозирование временных рядов, обработку естественного языка и генеративное глубокое обучение. Эти сложные приложения потребуют гораздо большего, чем просто архитектура `keras_model_`

`sequential()` и цикл `fit()` по умолчанию. Поэтому сначала превратим вас в эксперта по Keras! В этой главе представлен углубленный обзор ключевых навыков работы с API Keras: все, что вам понадобится для работы с расширенными вариантами использования глубокого обучения, с которыми вы столкнетесь далее.

## 7.1 Широкий спектр рабочих процессов Keras

Ключевой принцип, которым мы руководствовались при разработке API Keras, – *постепенное раскрытие сложности*. Мы стремились упростить начало работы, но при этом дать возможность работать с очень сложными задачами, требуя лишь постепенного обучения на каждом этапе. Простые варианты использования максимально доступны; при этом остается возможность создавать расширенные рабочие процессы с произвольной сложностью. Независимо от того, насколько нишевым и сложным является ваш проект, к нему должен вести четкий путь, основанный на знании более простых рабочих процессов. Это означает, что вы можете вырасти из новичка в эксперта и по-прежнему использовать одни и те же инструменты, только по-разному.

Таким образом, не существует единственного «настоящего» способа использования Keras. Напротив, Keras предлагает *широкий спектр рабочих процессов*, от очень простых до очень гибких. Существуют разные способы построения моделей Keras и разные способы их обучения, отвечающие разным потребностям. Поскольку все эти рабочие процессы основаны на общих API, таких как `Layer` и `Model`, компоненты любого рабочего процесса могут использоваться в другом рабочем процессе – все они могут взаимодействовать друг с другом.

## 7.2 Различные способы построения моделей Keras

Для построения моделей в Keras предназначены три API (рис. 7.1):

- *последовательный API* (Sequential API) – наиболее доступный из всех. По сути, это список, ограниченный простым набором слоев;
- *функциональный API* (Functional API), ориентированный на архитектуры графоподобных моделей. Он представляет собой нечто среднее между удобством использования и гибкостью и поэтому является наиболее часто используемым API для построения моделей;
- *подклассы моделей* – низкоуровневый вариант, когда вы пишете все сами с нуля. Это идеальный вариант, если вы хотите иметь

полный контроль над каждой мелочью. Однако у вас не будет доступа ко многим встроенным функциям Keras, и вы рискуете допустить ошибку.

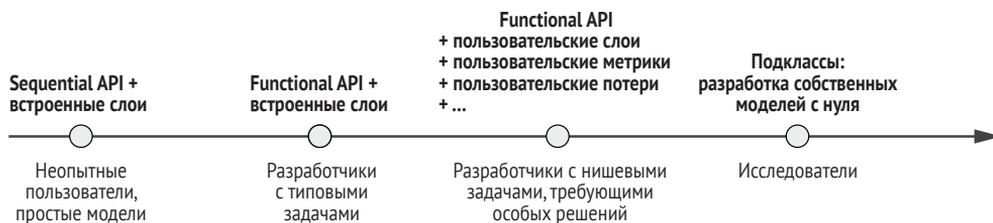


Рис. 7.1 Постепенное раскрытие сложности при построении модели

## 7.2.1 Sequential API

Самый простой способ построить модель Keras – использовать метод `keras_model_sequential()`, о котором вы уже знаете.

### Листинг 7.1 Использование метода `keras_model_sequential()`

```
library(keras)

model <- keras_model_sequential() %>%
  layer_dense(64, activation = "relu") %>%
  layer_dense(10, activation = "softmax")
```

Заметим, что ту же самую модель можно построить поэтапно с помощью оператора конвейера `%>%`.

### Листинг 7.2 Поэтапное построение модели с помощью Sequential API

```
model <- keras_model_sequential()
model %>% layer_dense(64, activation = "relu")
model %>% layer_dense(10, activation = "softmax")
```

Вы видели в главе 4, что построение слоев (то есть создание их весов) возможно только тогда, когда они вызываются в первый раз. Дело в том, что форма весов слоев зависит от формы их ввода: пока форма ввода неизвестна, их нельзя создать.

Таким образом, предыдущая модель Sequential не имеет весов (листинг 7.3) до тех пор, пока вы фактически не вызовете ее для некоторых данных или не вызовете ее метод `build()` с явной формой входных данных (листинг 7.4).

### Листинг 7.3 Модели, которые еще не построены, не имеют весов

```
model$weights ← На данный момент модель еще не построена
```

```
Error in py_get_attr_impl(x, name, silent):
ValueError: Weights for model sequential_1 have not yet been created. Weights
```

are created when the Model is first called on inputs or `build()` is called with an `input\_shape`.

#### Листинг 7.4 Первый вызов модели для ее построения

Построение модели – теперь модель будет ожидать образцы формы (3).

NA во входной форме сигнализирует о том, что размер пакета может быть любым

```
model$build(input_shape = shape(NA, 3))
str(model$weights)
```

← Теперь вы можете получить веса модели

List of 4

```
$ :<tf.Variable 'dense_2/kernel:0' shape=(3, 64) dtype=float32, numpy=...>
$ :<tf.Variable 'dense_2/bias:0' shape=(64) dtype=float32, numpy=...>
$ :<tf.Variable 'dense_3/kernel:0' shape=(64, 10) dtype=float32, numpy=...>
$ :<tf.Variable 'dense_3/bias:0' shape=(10) dtype=float32, numpy=...>
```

После того как модель построена, вы можете отобразить ее содержимое с помощью метода `print()`, который удобен для отладки.

#### Листинг 7.5 Использование метода `print()`

```
model
```

```
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 по умолчанию называется `sequential_1`. В Keras вы можете присваивать имена чему угодно – каждой модели, каждому слою.

#### Листинг 7.6 Именованние моделей и слоев с помощью аргумента `name`

```
model <- keras_model_sequential(name = "my_example_model")
model %>% layer_dense(64, activation = "relu", name = "my_first_layer")
model %>% layer_dense(10, activation = "softmax", name = "my_last_layer")
model$build(shape(NA, 3))
model
```

```
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 поэтапном построении последовательной модели полезно иметь возможность распечатать сводку о том, как выглядит текущая модель после добавления каждого слоя. Но вы не можете распечатать сводку, пока модель не построена! На самом деле есть способ построить вашу последовательную модель «на лету»: просто заранее объявите форму входных данных модели. Вы можете сделать это, передав параметр `input_shape` в `keras_model_sequential()`.

### Листинг 7.7 Предварительное указание входной формы вашей модели

```
model <-
  keras_model_sequential(input_shape = c(3)) %>% ←
  layer_dense(64, activation = "relu")
```

Предоставьте `input_shape`, чтобы объявить форму входных данных.  
Обратите внимание, что аргумент `shape` должен быть формой  
каждого образца, а не одного пакета

Теперь вы можете использовать метод `print()`, чтобы следить за тем, как изменяется форма выхода вашей модели при добавлении дополнительных слоев:

```
model
```

```
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 %>% layer_dense(10, activation = "softmax")
```

```
model
```

```
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 рабочий процесс отладки при работе со слоями, которые сложным образом преобразуют свои входные данные. К ним относятся сверточные слои, о которых вы узнаете в главе 8.

## 7.2.2 Functional API

Последовательная модель проста в использовании, но ее применимость чрезвычайно ограничена: она может выражать модели только с одним входом и одним выходом, последовательно применяя один слой за другим. На практике довольно часто встречаются модели с несколькими входами (скажем, изображение и его метаданные), несколькими выходами (разные вещи, которые вы хотите спрогнозировать относительно данных) или с нелинейной топологией.

В таких случаях вы должны построить свою модель с помощью Functional API. Этот API используют почти все модели Keras, с которыми вы столкнетесь в реальной жизни. Работать с Functional API очень увлекательно и круто – это похоже на игру с кубиками LEGO.

### ПРОСТОЙ ПРИМЕР

Давайте начнем с простой модели: последовательности из двух слов, которую мы использовали в предыдущем разделе. Версия этой модели для Functional API представлена в листинге 7.8.

#### Листинг 7.8 Простая модель с двумя слоями Dense для Functional API

```
inputs <- layer_input(shape = c(3), name = "my_input")
features <- inputs %>% layer_dense(64, activation = "relu")
outputs <- features %>% layer_dense(10, activation = "softmax")
model <- keras_model(inputs = inputs, outputs = outputs)
```

Мы начали с объявления `layer_input()` (вы можете присваивать имена объектам ввода, как и всему остальному):

```
inputs <- layer_input(shape = c(3), name = "my_input")
```

Этот входной объект содержит информацию о форме и типе данных, которые будет обрабатывать модель:

```
inputs$shape
TensorShape([None, 3])
```

← Модель будет обрабатывать пакеты, в которых каждый образец имеет форму (3). Количество образцов в партии является переменным (обозначается размером партии None)

`inputs$dtype` ← Пакеты относятся к типу данных `float32`

```
tf.float32
```

Мы называем такой объект *символическим тензором*. Он не содержит никаких фактических данных, но кодирует спецификации фактических тензоров данных, которые модель увидит при ее использовании. Он просто *обозначает* будущие тензоры данных.

Далее мы создаем слой и связываем его с вводом:

```
features <- inputs %>% layer_dense(64, activation = "relu")
```

В Functional API передача символического тензора в конструктор слоя вызывает метод `call()` слоя. По сути, происходит вот что:

```
layer_instance <- layer_dense(units = 64, activation = "relu")
features <- layer_instance(inputs)
```

В этом заключается отличие от Sequential API, где создание слоя при помощи `model %>% layer_dense()` означает следующее:

```
layer_instance <- layer_dense(units = 64, activation = "relu")
model$add(layer_instance)
```

Все слои Keras можно вызывать как с реальными тензорами данных, так и с символическими тензорами. В последнем случае они возвращают новый символический тензор с обновленной информацией о форме и типе данных:

```
features$shape
```

```
TensorShape([None, 64])
```

Обратите внимание, что символические тензоры работают почти со всеми теми же универсальными методами R, что и обычные тензоры. Например, вы можете получить форму в виде целочисленного вектора R:

```
dim(features)
```

```
[1] NA 64
```

Далее мы создаем экземпляр модели, указав ее входные и выходные данные в конструкторе `keras_model()`:

```
outputs <- layer_dense(features, 10, activation = "softmax")
model <- keras_model(inputs = inputs, outputs = outputs)
```

Вот краткая сводка нашей модели:

```
model
```

```
Model: "model_1"
```

---

Layer (type)	Output Shape	Param #
--------------	--------------	---------

```

=====
my_input (InputLayer)          [(None, 3)]          0
dense_8 (Dense)                (None, 64)          256
dense_9 (Dense)                (None, 10)          650
=====
Total params: 906
Trainable params: 906
Non-trainable params: 0

```

## Модели с несколькими входами и выходами

В отличие от этой демонстрационной модели, большинство моделей глубокого обучения выглядят не как списки, а как графы. Например, они могут иметь несколько входов или несколько выходов. Именно для таких моделей лучше всего подходит Functional API.

Предположим, вы создаете систему для ранжирования заявок в службу поддержки по приоритету и отправки их в соответствующий отдел. Ваша модель имеет три входа:

- заголовок заявки (текстовый вход);
- тело заявки (текстовый вход);
- любые теги, добавленные пользователем (категориальный вход, здесь предполагается прямое унитарное кодирование).

Мы можем кодировать текстовые входные данные как массивы единиц и нулей размера `vocabulary_size` (см. главу 11 для получения подробной информации о методах кодирования текста). Эта модель также имеет два выхода:

- оценка приоритета заявки, скаляр от 0 до 1 (сигмоидная функция);
- отдел, который должен обрабатывать заявку (softmax по набору отделов).

Благодаря использованию Functional API такая модель может состоять всего из нескольких строк (листинг 7.9).

**Листинг 7.9** Модель с несколькими входами и несколькими выходами на основе Functional API

```

vocabulary_size <- 10000
num_tags <- 100
num_departments <- 4

title <- layer_input(shape = c(vocabulary_size), name = "title")
text_body <- layer_input(shape = c(vocabulary_size), name = "text_body")
tags <- layer_input(shape = c(num_tags), name = "tags")

features <-
  layer_concatenate(list(title, text_body, tags)) %>%
  layer_dense(64, activation = "relu")

```

Определение входных данных модели

Объединение входных признаков в один тензор путем конкатенации

Применение промежуточного слоя, чтобы рекомбинировать входные объекты в более богатые представления

```

priority <- features %>%
  layer_dense(1, activation = "sigmoid", name = "priority")
department <- features %>%
  layer_dense(num_departments, activation = "softmax", name = "department")

model <- keras_model(
  inputs = list(title, text_body, tags),
  outputs = list(priority, department)

```

← Определение выходных данных модели

← Создание модели путем указания ее входных и выходных данных

Как видите, Functional API – это простой, похожий на LEGO, но очень гибкий способ определения произвольных графов слоев.

## ОБУЧЕНИЕ МОДЕЛИ С НЕСКОЛЬКИМИ ВХОДАМИ И НЕСКОЛЬКИМИ ВЫХОДАМИ

Вы можете обучать эту модель почти так же, как и последовательную модель, вызывая метод `fit()` со списками входных и выходных данных. Эти списки данных должны следовать в том же порядке, что и входные данные, которые вы передали конструктору `keras_model()`.

### Листинг 7.10 Обучение модели путем предоставления списков входных и целевых массивов

```

num_samples <- 1280

random_uniform_array <- function(dim)
  array(runif(prod(dim)), dim)

random_vectorized_array <- function(dim)
  array(sample(0:1, prod(dim), replace = TRUE), dim)

title_data <- random_vectorized_array(c(num_samples, vocabulary_size))
text_body_data <- random_vectorized_array(c(num_samples, vocabulary_size))
tags_data <- random_vectorized_array(c(num_samples, num_tags))

priority_data <- random_vectorized_array(c(num_samples, 1))
department_data <- random_vectorized_array(c(num_samples, num_departments))

model %>% compile(
  optimizer = "rmsprop",
  loss = c("mean_squared_error", "categorical_crossentropy"),
  metrics = c("mean_absolute_error", "accuracy")
)

model %>% fit(
  x = list(title_data, text_body_data, tags_data),
  y = list(priority_data, department_data),
  epochs = 1
)

model %>% evaluate(x = list(title_data, text_body_data, tags_data),

```

Имитация входных данных

Имитация выходных данных

```

y = list(priority_data, department_data))

          loss                priority_loss
39.8363457                0.5007812
department_loss    priority_mean_absolute_error
39.3355637                0.5007812
priority_accuracy    department_mean_absolute_error
0.4992188                0.5046247
department_accuracy
0.2351563

```

```

c(priority_preds, department_preds) %<-% {
  model %>% predict(list(title_data, text_body_data, tags_data))
}

```

← Чтобы использовать %<-% и %>% в одном выражении, вам нужно обернуть последовательность каналов в скобки {} или () для переопределения приоритета оператора по умолчанию

Если вы не хотите полагаться на порядок ввода (например, потому что у вас много входов или выходов), вы также можете использовать имена, которые вы дали `input_shape` и выходным слоям, и передавать данные через именованный список.

**ВАЖНО!** При использовании именованных списков не гарантируется сохранение порядка в списке. Обязательно отслеживайте элементы *или* по порядку, *или* по имени, но не по комбинации того и другого.

#### Листинг 7.11 Обучение модели путем предоставления именованных списков входных и выходных массивов

```

model %>%
  compile(optimizer = "rmsprop",
          loss = c(priority = "mean_squared_error",
                  department = "categorical_crossentropy"),
          metrics = c(priority = "mean_absolute_error",
                      department = "accuracy"))

model %>%
  fit(list(title = title_data,
          text_body = text_body_data,
          tags = tags_data),
      list(priority = priority_data,
          department = department_data), epochs = 1)

model %>%
  evaluate(list(title = title_data,
              text_body = text_body_data,
              tags = tags_data),
          list(priority = priority_data,
              department = department_data))

```

```

        loss                priority_loss
35.7301750                0.5007812
department_loss          priority_mean_absolute_error
35.2293930                0.5007812
department_accuracy
0.1320312

```

```

c(priority_preds, department_preds) %<-%
predict(model, list(title = title_data,
                    text_body = text_body_data,
                    tags = tags_data))

```

## Сила FUNCTIONAL API: доступ к СВЯЗНОСТИ СЛОЕВ

*Функциональная модель* – это явная графовая структура данных. Она позволяет смотреть, как связаны слои, и повторно использовать предыдущие узлы графа (которые являются выходными данными слоя) как часть новых моделей. Это также хорошо соответствует ментальной модели, которую большинство исследователей используют, представляя глубокую нейронную сеть, а именно *графу слоев*. Благодаря такому представлению перед нами открываются два важных варианта использования: визуализация модели и извлечение признаков.

Давайте визуализируем связность модели, которую мы только что определили (*топологию* модели). Вы можете визуализировать функциональную модель в виде графа с помощью метода `plot()` (рис. 7.2):

```
plot(model)
```

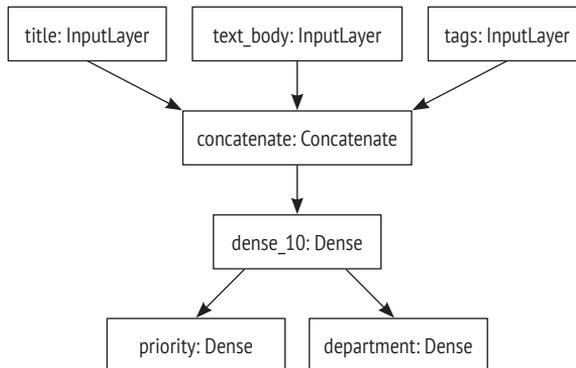


Рис. 7.2 Граф, сгенерированный методом `plot(model)` в нашей модели классификатора билетов

Вы можете добавить к этому графу входные и выходные формы каждого слоя в модели, что может быть полезно во время отладки (рис. 7.3):

```
plot(model, show_shapes = TRUE)
```

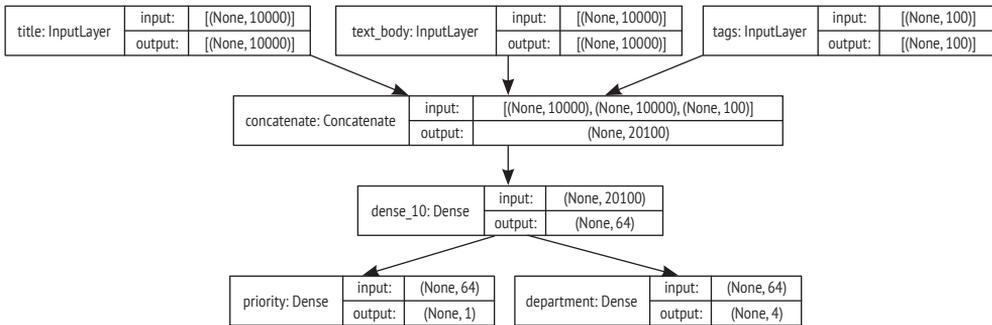


Рис. 7.3 Граф модели с дополнительной информацией о форме

«None» в формах тензоров представляет размер пакета: данная модель допускает пакеты любого размера.

Доступ к связности слоев также означает, что вы можете просматривать и повторно использовать отдельные узлы (вызовы слоев) в графе. Свойство модели `model.layers` предоставляет список слоев, из которых состоит модель, и для каждого слоя вы можете запросить `Layer.Input` и `Layer.Output`.

#### Листинг 7.12 Получение входных или выходных данных слоя в функциональной модели

```
str(model.layers)
```

```
List of 7
```

```
$ :<keras.engine.input_layer.InputLayer object at 0x7fc962da63a0>
$ :<keras.engine.input_layer.InputLayer object at 0x7fc962da6430>
$ :<keras.engine.input_layer.InputLayer object at 0x7fc962da68e0>
$ :<keras.layers.merge.Concatenate object at 0x7fc962d2e130>
$ :<keras.layers.core.dense.Dense object at 0x7fc962da6c40>
$ :<keras.layers.core.dense.Dense object at 0x7fc962da6340>
$ :<keras.layers.core.dense.Dense object at 0x7fc962d331f0>
```

```
str(model.layers[[4]]$input)
```

```
List of 3
```

```
$ :<KerasTensor: shape=(None, 10000) dtype=float32 (created by layer
➔ 'title')>
$ :<KerasTensor: shape=(None, 10000) dtype=float32 (created by layer
➔ 'text_body')>
$ :<KerasTensor: shape=(None, 100) dtype=float32 (created by layer
➔ 'tags')>
```

```
str(model.layers[[4]]$output)
```

```
<KerasTensor: shape=(None, 20100) dtype=float32 (created by layer
➔ 'concatenate')>
```

Это позволяет вам выполнять *извлечение признаков*, создавая модели, которые повторно используют промежуточные признаки из другой модели.

Допустим, вы хотите добавить еще один выход к предыдущей модели – хотите оценить, сколько времени потребуется на решение данной заявки, – своего рода рейтинг сложности. Это можно сделать с помощью слоя классификации по трем категориям: «легкая», «средняя» и «сложная». Вам не нужно создавать и обучать новую модель с нуля. Вы можете начать с промежуточных функций вашей предыдущей модели, потому что у вас есть к ним доступ, как показано в листинге 7.13.

**Листинг 7.13 Создание новой модели путем повторного использования выходных данных промежуточного слоя**

```

layer[[5]] – наш промежуточный полносвязный слой.
Вы также можете получить слой по имени с помощью get_layer()

features <- model$layers[[5]]$output
difficulty <- features %>%
  layer_dense(3, activation = "softmax", name = "difficulty")

new_model <- keras_model(
  inputs = list(title, text_body, tags),
  outputs = list(priority, department, difficulty)
)
    
```

Построим граф нашей новой модели (рис. 7.4):

```
plot(new_model, show_shapes = TRUE)
```

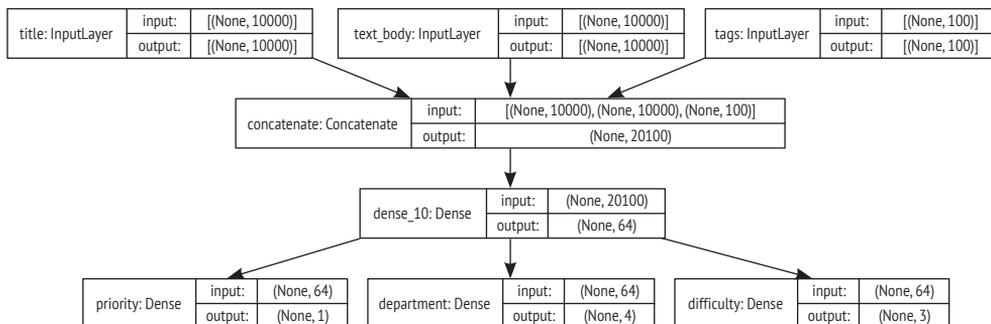


Рис. 7.4 Граф новой модели: обновленный классификатор заявок

### 7.2.3 Создание подкласса класса Model

Последний способ построения модели, о котором вы должны знать, является наиболее продвинутым: создание подклассов класса Model. В главе 3 вы узнали, как использовать new\_layer\_class() для созда-

ния подкласса класса `Layer` и создания пользовательских слоев. Использование `new_model_class()` для подкласса класса `Model` очень похоже:

- в методе `initialize()` определите слои, которые будет использовать модель;
- в методе `call()` определите прямой проход модели, повторно используя ранее созданные слои;
- создайте экземпляр своего подкласса и вызовите его для данных, чтобы создать его веса.

## МОДИФИКАЦИЯ ПРЕДЫДУЩЕГО ПРИМЕРА С ПОДКЛАССОВОЙ МОДЕЛЮ

В качестве простого примера повторно реализуем модель управления обращениями в службу поддержки клиентов, используя `new_model_class()` для определения подкласса `Model` (листинг 7.14).

**Листинг 7.14 Простая подклассовая модель**

```
CustomerTicketModel <- new_model_class(
  classname = "CustomerTicketModel",

  initialize = function(num_departments) {
    super$initialize()
    self$concat_layer <- layer_concatenate()
    self$mixing_layer <-
      layer_dense(units = 64, activation = "relu")
    self$priority_scorer <-
      layer_dense(units = 1, activation = "sigmoid")
    self$department_classifier <-
      layer_dense(units = num_departments, activation = "softmax")
  },

  call = function(inputs) {
    title <- inputs$title
    text_body <- inputs$text_body
    tags <- inputs$tags

    features <- list(title, text_body, tags) %>%
      self$concat_layer() %>%
      self$mixing_layer()
    priority <- self$priority_scorer(features)
    department <- self$department_classifier(features)
    list(priority, department)
  }
)
```

Не забудьте вызвать функцию `super$initialize()`!

Определите прямой проход в методе `call()`

Определите подслои в конструкторе. Обратите внимание, что здесь мы указываем имя, чтобы получить экземпляр слоя

Для входных данных мы предоставим модели именованный список

Мы реализовали простую базовую версию класса `Model` в главе 3. Главное, о чем следует помнить, – мы определяем пользовательский класс, нашу модель, которая является подклассом `Model`. Класс `Model`

предоставляет множество методов и возможностей. В следующих разделах вы увидите, как ими можно воспользоваться.

Определив модель, вы можете создать ее экземпляр. Обратите внимание, что по аналогии с подклассом `Layer` веса будут созданы только при первом вызове экземпляра для каких-либо данных:

```
model <- CustomerTicketModel(num_departments = 4)

c(priority, department) %<-% model(list(title = title_data,
                                       text_body = text_body_data,
                                       tags = tags_data))
```

Модели, по сути, относятся к типу `Layer`. Это означает, что вы можете легко добавить возможность элегантной упаковки модели в конвейер `%>%` с помощью `create_layer_wrapper()`, например:

```
inputs <- list(title = title_data,
               text_body = text_body_data,
               tags = tags_data)

layer_customer_ticket_model <- create_layer_wrapper(CustomerTicketModel)

outputs <- inputs %>%
  layer_customer_ticket_model(num_departments = 4)
c(priority, department) %<-% outputs
```

До сих пор все выглядело очень похоже на подклассы класса `Layer`, с рабочим процессом которого вы познакомились в главе 3. В чем же тогда разница между подклассами `Layer` и подклассами `Model`? Все просто: «слой» – это строительный блок, который вы используете для создания моделей, а «модель» – это объект верхнего уровня, который вы фактически будете обучать, экспортировать для логического вывода и т. д. Если коротко, класс `Model` имеет методы `fit()`, `evaluate()` и `predict()`. У класса `Layer` их нет. В остальном эти два класса практически идентичны. (Еще одно отличие состоит в том, что вы можете сохранить модель в файл на диске, о чем мы расскажем позже.) Вы можете скомпилировать и обучить подкласс `Model` точно так же, как последовательную или функциональную модель:

```
model %>%
  compile(optimizer = "rmsprop",
          loss = c("mean_squared_error",
                  "categorical_crossentropy"),
          metrics = c("mean_absolute_error", "accuracy"))

x <- list(title = title_data,
          text_body = text_body_data,
```

Структура входных данных должна точно соответствовать структуре, ожидаемой методом `call()` – в данном случае именованный список с элементами `title`, `text_body` и тегами. (Помните, порядок списка игнорируется при наличии сопоставления по именам!)

```

tags = tags_data)
y <- list(priority_data, department_data)
model %>% fit(x, y, epochs = 1)
model %>% evaluate(x, y)

```

Структура целевых данных должна точно соответствовать тому, что возвращает метод `call()` – в данном случае список из двух элементов

```

      loss      output_1_loss
24.02798843      0.50078124
output_2_loss output_1_mean_absolute_error
23.52721024      0.50078124
output_1_accuracy output_2_mean_absolute_error
0.49921876      0.50347400
output_2_accuracy
0.06328125

```

```

c(priority_preds, department_preds) %<-% {
  model %>% predict(x)
}

```

Рабочий процесс создания подклассов `Model` – наиболее гибкий способ построения модели. Это позволяет вам строить модели, которые нельзя выразить в виде ориентированных ациклических графов слоев, – представьте себе, например, модель, в которой метод `call()` использует слои внутри цикла `for` или даже вызывает их рекурсивно. Все возможно – вы здесь главный.

## ВНИМАНИЕ: НА ЧТО НЕ СПОСОБНЫ ПОДКЛАССОВЫЕ МОДЕЛИ

За эту свободу приходится платить: в подклассовых моделях вы отвечаете за большую часть логики модели, а это означает, что ваша потенциальная поверхность ошибок намного больше. В результате у вас будет больше работы по отладке. Здесь вы разрабатываете новый объект класса, а не просто соединяете кубики LEGO между собой.

Функциональные и подклассовые модели также существенно различаются по своей природе. Функциональная модель – это явная структура данных, т. е. граф слоев, который вы можете просматривать, проверять и изменять. Подклассовая модель – это набор кода R, т. е. класс с методом `call()`, который является функцией R. В этом источник гибкости рабочего процесса создания подклассов – вы можете закодировать любую функциональность, которая вам нравится, – но это вводит новые ограничения. Например, поскольку способ соединения слоев друг с другом скрыт внутри тела метода `call()`, вы не можете получить доступ к этой информации. Вызов `summary()` не отобразит связность слоев, и вы не сможете построить топологию модели с помощью `plot()`. Аналогично, если у вас есть подклассовая модель, вы не можете получить доступ к узлам графа слоев для извлечения признаков, потому что графа просто нет. Прямой проход готовой модели становится полным черным ящиком.

## 7.2.4 Смешивание и сочетание разных компонентов

Важно отметить, что выбор одного из вариантов – последовательной модели, Functional API или подкласса Model – не исключает другие. Все модели в Keras API могут беспрепятственно взаимодействовать друг с другом, будь то последовательные, функциональные или подклассовые модели, написанные с нуля. Все они являются частью одного и того же спектра рабочих процессов. Например, вы можете использовать подклассовый слой или модель в функциональной модели (листинг 7.15).

**Листинг 7.15** Создание функциональной модели, включающей подклассовую модель

```
ClassifierModel <- new_model_class(
  classname = "Classifier",
  initialize = function(num_classes = 2) {
    super$initialize()
    if (num_classes == 2) {
      num_units <- 1
      activation <- "sigmoid"
    } else {
      num_units <- num_classes
      activation <- "softmax"
    }
    self$dense <- layer_dense(units = num_units, activation = activation)
  },
  call = function(inputs)
    self$dense(inputs)
)
```

И наоборот, вы можете использовать функциональную модель как часть подклассового слоя или модели (листинг 7.16).

**Листинг 7.16** Создание подклассовой модели, включающей функциональную модель

```
inputs <- layer_input(shape = c(64))
outputs <- inputs %>% layer_dense(1, activation = "sigmoid")
binary_classifier <- keras_model(inputs = inputs, outputs = outputs)

MyModel <- new_model_class(
  classname = "MyModel",
  initialize = function(num_classes = 2) {
    super$initialize()
    self$dense <- layer_dense(units = 64, activation = "relu")
    self$classifier <- binary_classifier
  },
  call = function(inputs) {
    inputs %>%
```

```

        self$dense() %>%
        self$classifier()
    }
)

model <- MyModel()

```

## 7.2.5 Используйте правильные инструменты

Вы познакомились с полным спектром рабочих процессов для построения моделей Keras, от простейшей последовательной до самой сложной подклассовой модели. У каждого варианта есть свои плюсы и минусы – выберите тот, который наиболее подходит для вашей работы.

В целом Functional API предлагает довольно хороший компромисс между простотой использования и гибкостью. Он также дает вам прямой доступ к связности слоев, что очень полезно для таких случаев использования, как построение модели или извлечение признаков. Если ваша модель может быть выражена в виде ориентированного ациклического графа слоев, я рекомендую использовать Functional API, а не подклассовую модель.

В дальнейшем все примеры в этой книге будут основаны на Functional API просто потому, что все модели, с которыми мы будем работать, могут быть представлены в виде графов слоев. Однако мы будем часто использовать подклассовые слои, задействуя `new_layer_class()`. В общем, использование функциональных моделей, включающих подклассы, обеспечивает наилучшее сочетание: высокую гибкость разработки при сохранении преимуществ Functional API.

## 7.3 Использование встроенных циклов обучения и оценки

Принцип постепенного раскрытия сложности – наличие набора рабочих процессов, позволяющего постепенно переходить от простейших моделей к максимально гибким и сложным, – также применим к обучению моделей. Keras предоставляет вам различные рабочие процессы для обучения моделей. Они могут быть простыми, как вызов `fit()` для ваших данных, или сложными, как написание нового алгоритма обучения с нуля.

Вы уже знакомы с рабочим процессом `compile()`, `fit()`, `evaluate()`, `predict()`. В качестве напоминания взгляните на следующий список.

**Листинг 7.17** Стандартный рабочий процесс: `compile()`, `fit()`, `evaluate()`, `predict()`

```

get_mnist_model <- function() {
    
    Создайте модель (мы выделяем ее в отдельную функцию,
    чтобы использовать ее позже)
}

```

```

inputs <- layer_input(shape = c(28 * 28))
outputs <- inputs %>%
  layer_dense(512, activation = "relu") %>%
  layer_dropout(0.5) %>%
  layer_dense(10, activation = "softmax")

keras_model(inputs, outputs)
}
c(c(images, labels), c(test_images, test_labels)) %<- %> ←
dataset_mnist()

```

Загрузите свои данные,  
зарезервировав некоторую  
часть для проверки

```

images <- array_reshape(images, c(-1, 28 * 28)) / 255
test_images <- array_reshape(test_images, c(-1, 28 * 28)) / 255

```

```

val_idx <- seq(10000)
val_images <- images[val_idx, ]
val_labels <- labels[val_idx]
train_images <- images[-val_idx, ]
train_labels <- labels[-val_idx]

```

Скомпилируйте модель, указав  
ее оптимизатор, функцию потерь  
для минимизации и показатели  
для мониторинга

```

model <- get_mnist_model()
model %>% compile(optimizer = "rmsprop", ←
  loss = "sparse_categorical_crossentropy",
  metrics = "accuracy")
model %>% fit(train_images, train_labels,
  epochs = 3,
  validation_data = list(val_images, val_labels)) ←

```

```

test_metrics <- model %>% evaluate(test_images,
  test_labels)

```

```

predictions <- model %>% predict(test_images)

```

Используйте predict() для вычисления  
вероятностей классов новых данных

Используйте evaluate() для вычисления  
потерь и метрик на новых данных

Используйте fit() для обучения модели,  
при необходимости предоставляя данные проверки  
для мониторинга точности на незнакомых данных

Есть несколько способов настроить этот простой рабочий процесс под свои потребности:

- указать свои собственные метрики;
- передать методу fit() *обратные вызовы*, чтобы запланировать действия, которые будут выполняться в определенные моменты во время обучения.

Далее мы рассмотрим эти способы более подробно.

### 7.3.1 Разработка собственных метрик

Метрики являются ключом к измерению качества вашей модели, в частности к измерению разницы между точностью на обучающих и контрольных данных. Наиболее популярные метрики для классификации и регрессии уже являются частью пакета Keras, все они

начинаются с префикса `metric_`, и большую часть времени вы будете использовать именно их. Но если вы строите необычную модель, вам нужно будет разработать свои собственные метрики. Это просто!

Метрика Keras является подклассом класса `Metric`. Как и слои, метрика имеет внутреннее состояние, хранящееся в переменных TensorFlow. В отличие от слоев, эти переменные не обновляются обратным распространением, поэтому вам придется писать логику обновления состояния самостоятельно; она работает в методе `update_state()`. Например, в листинге 7.18 представлена простая пользовательская метрика, которая измеряет *среднеквадратичную ошибку* (root mean squared error, RMSE).

### Листинг 7.18 Реализация пользовательской метрики путем создания подкласса класса `Metric`

Здесь мы будем использовать функции модуля `tf`

Определите переменные состояния в конструкторе. Как и для слоев, у вас есть доступ к методу `add_weight()`

```
library(tensorflow)

metric_root_mean_squared_error <- new_metric_class(
  classname = "RootMeanSquaredError",

  initialize = function(name = "rmse", ...) {
    super$initialize(name = name, ...)
    self$mse_sum <- self$add_weight(name = "mse_sum",
                                   initializer = "zeros",
                                   dtype = "float32")
    self$total_samples <- self$add_weight(name = "total_samples",
                                          initializer = "zeros",
                                          dtype = "int32")
  },

  update_state = function(y_true, y_pred, sample_weight = NULL) {
    num_samples <- tf$shape(y_pred)[1]
    num_features <- tf$shape(y_pred)[2]

    y_true <- tf$one_hot(y_true, depth = num_features)

    mse <- sum((y_true - y_pred) ^ 2)
    self$mse_sum$assign_add(mse)
    self$total_samples$assign_add(num_samples)
  },

  result = function() {
    tf$reduce_sum(tf$square(tf$subtract(y_true, y_pred)))
  }
)
```

Определите новый класс, который является подклассом базового класса `Metric`

Чтобы соответствовать нашей модели MNIST, мы ожидаем категориальные прогнозы и целочисленные метки

Учтите, что функции модуля `tf` используют отсчет с 0. Значение 0 в `y_true` помещает 1 в первую позицию унитарного вектора

Реализуйте логику обновления состояния в `update_state()`. Аргумент `y_true` — это цели (или метки) для одного пакета, а `y_pred` представляет соответствующие прогнозы модели. Вы можете игнорировать аргумент `sample_weight` — мы не будем его здесь использовать

```

    sqrt(self$mse_sum /
          tf$cast(self$total_samples, "float32"))
  },
  reset_state = function() {
    self$mse_sum$assign(0)
    self$total_samples$assign(0L)
  }
)

```

Приведите total\_samples к типу mse\_sum

Обратите внимание, что в `update_state()` мы используем `tf$shape(y_pred)` вместо `y_pred$shape`. Метод `tf$shape()` возвращает форму как `tf.Tensor`, а не `tf.TensorShape`, как `y_pred$shape`. Метод `tf$shape()` позволяет `tf_function()` скомпилировать функцию, способную работать с тензорами с неопределенными формами, такими как наши входные данные здесь, которые имеют неопределенное измерение пакетов. Вскоре вы узнаете больше о `tf_function()`.

Далее мы используем метод `result()` для возврата текущего значения метрики:

```

result = function()
  sqrt(self$mse_sum /
        tf$cast(self$total_samples, "float32"))

```

Кроме того, нам также необходимо предоставить способ сброса состояния метрики без необходимости повторного создания – это позволяет использовать одни и те же объекты метрики в разные периоды обучения или как при обучении, так и при оценке. Это делается с помощью метода `reset_state()`:

```

reset_state = function() {
  self$mse_sum$assign(0)
  self$total_samples$assign(0L)
}

```

Обратите внимание, что мы передаем целое число, потому что total\_samples относится к целочисленному типу

Пользовательские метрики можно использовать так же, как и встроенные. Давайте протестируем нашу новую метрику:

```

model <- get_mnist_model()
model %>%
  compile(optimizer = "rmsprop",
           loss = "sparse_categorical_crossentropy",
           metrics = list("accuracy", metric_root_mean_squared_error()))
model %>%
  fit(train_images, train_labels,
       epochs = 3,
       validation_data = list(val_images, val_labels))
test_metrics <- model %>% evaluate(test_images, test_labels)

```

Теперь вы можете видеть прогресс выполнения метода `fit()` в виде изменений RMSE вашей модели.

### 7.3.2 Использование обратных вызовов

Запуск обучающего прогона на большом наборе данных для десятков эпох с использованием метода `fit()` напоминает запуск бумажного самолетика: после броска вы не можете влиять на его траекторию или точку приземления. Если вы хотите избежать плохих результатов (и, следовательно, потраченных напрасно бумажных самолетов), разумнее запускать не бумажный самолетик, а дрон, способный воспринимать окружающую среду, отправлять данные своему оператору и автоматически принимать решения по управлению на основе его текущего состояния. API *обратных вызовов* Keras поможет вам превратить вызов `fit(model)` из бумажного самолетика в умный автономный дрон, обладающий способностью к самоанализу и динамичному поведению.

Фактически обратный вызов – это объект (экземпляр класса, реализующий определенные методы), который передается модели при вызове `fit()` и вызывается моделью в различные моменты во время обучения. У него есть доступ ко всем доступным данным о состоянии модели и ее точности, и он может предпринимать ряд действий: прерывать обучение, сохранять модель, загружать другой набор весов или иным образом изменять состояние модели. Вот несколько примеров того, как вы можете использовать обратные вызовы:

- *контрольные точки модели* – сохранение текущего состояния модели в разные моменты во время обучения;
- *ранняя остановка* – прерывание обучения, когда потеря проверки больше не улучшается (и, конечно же, сохранение лучшей модели, полученной во время обучения);
- *динамическая настройка значений определенных параметров во время обучения*, например скорость обучения оптимизатора;
- *регистрация показателей обучения и проверки во время обучения или визуализация представлений, изученных моделью, по мере их обновления* – отслеживание выполнения метода `fit()`, с которым вы знакомы, на самом деле является обратным вызовом!

В состав пакета Keras входят различные встроенные обратные вызовы (это не исчерпывающий список):

```
callback_model_checkpoint()  
callback_early_stopping()  
callback_learning_rate_scheduler()  
callback_reduce_lr_on_plateau()  
callback_csv_logger()  
...
```

Давайте рассмотрим два из них – `callback_early_stop()` и `callback_model_checkpoint()`, – чтобы вы получили представление о том, как их используют.

## ОБРАТНЫЕ ВЫЗОВЫ `CALLBACK_EARLY_STOP()` И `CALLBACK_MODEL_CHECKPOINT()`

Когда вы обучаете модель, есть много вещей, которые вы не можете предсказать с самого начала. В частности, вы не можете сказать, сколько эпох потребуется для достижения оптимальной потери при проверке. В наших примерах до сих пор применялась стратегия обучения в течение достаточного количества эпох, когда мы достигали переобучения в течение первого прогона, чтобы определить правильное количество эпох, а затем запускали новый прогон обучения с нуля, используя это количество. Разумеется, такой подход расточителен. Гораздо лучший выход – немедленно прекратить обучение, когда вы заметите, что потеря на проверочных данных перестала убывать. Этого можно добиться с помощью обратного вызова `callback_early_stopping()`.

Обратный вызов с ранней остановкой прерывает обучение, как только отслеживаемая целевая метрика перестает улучшаться в течение фиксированного количества эпох. Например, этот обратный вызов позволяет вам прервать обучение, как только вы начнете переобучение, избегая, таким образом, необходимости в повторном обучении вашей модели на меньшем количестве эпох. Этот обратный вызов обычно используется в сочетании с `callback_model_checkpoint()`, который позволяет постоянно сохранять модель во время обучения (и, при желании, сохранять только текущую наилучшую модель – версию модели, которая достигла наилучших показателей в конце эпохи обучения).

### Листинг 7.19 Использование аргумента `callbacks` в методе `fit()`

```

callbacks_list <- list(
  callback_early_stopping(
    monitor = "val_accuracy", patience = 2),
  callback_model_checkpoint(
    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,

```

Прерываем обучение, когда точность проверки перестанет улучшаться в течение двух эпох

Сохраняем текущие веса после каждой эпохи

Путь к файлу целевой модели

Эти два аргумента означают, что вы не будете перезаписывать файл модели до тех пор, пока `val_loss` не улучшится, что позволит вам сохранить лучшую модель, наблюдаемую во время обучения

Вы следите за точностью, поэтому она должна быть частью показателей модели

Обратные вызовы передаются модели через аргумент `callbacks` в `fit()`, который принимает список обратных вызовов. Вы можете передать любое количество обратных вызовов

Поскольку обратный вызов будет отслеживать потери и точность на проверочных данных, вам необходимо передать `validation_data` в вызов функции `fit()`

```
callbacks = callbacks_list,
validation_data = list(val_images, val_labels))
```

Кстати, вы всегда можете сохранить модели вручную после обучения – просто вызовите `save_model_tf(model, 'my_checkpoint_path')`. Чтобы загрузить сохраненную модель, воспользуйтесь следующей командой:

```
model <- load_model_tf("checkpoint_path.keras")
```

### 7.3.3 Разработка собственных обратных вызовов

Если вам нужно выполнить определенное действие во время обучения и для него не предусмотрен встроенный обратный вызов, разработайте собственное решение. Обратные вызовы реализуются в виде подкласса класса `Callback` Keras с помощью `new_callback_class()`. Вы можете реализовать любое количество следующих методов с интуитивно понятными именами, которые вызываются в различные моменты во время обучения:

Вызывается в начале каждой эпохи	→	<code>on_epoch_begin(epoch, logs)</code>	←	Вызывается в конце каждой эпохи
	→	<code>on_epoch_end(epoch, logs)</code>	←	Вызывается перед началом обработки каждого пакета
	→	<code>on_batch_begin(batch, logs)</code>	←	
	→	<code>on_batch_end(batch, logs)</code>	←	
	→	<code>on_train_begin(logs)</code>	←	Вызывается после завершения обработки каждого пакета
	→	<code>on_train_end(logs)</code>	←	Вызывается при запуске обучения
Вызывается при завершении обучения	→		←	

Все эти методы вызываются с аргументом `logs`, который представляет собой именованный список, содержащий информацию о предыдущем пакете, эпохе или прогоне обучения – метрики обучения и проверки и т. д. Методы `on_epoch_*` и `on_batch_*` также принимают индекс эпохи или пакета в качестве первого аргумента (целое число).

Рассмотрим простой пример (листинг 7.20), который сохраняет список значений потерь для каждого пакета во время обучения и строит график этих значений в конце каждой эпохи.

#### Листинг 7.20 Создание пользовательского обратного вызова путем создания подкласса класса обратного вызова

```
callback_plot_per_batch_loss_history <- new_callback_class(
  classname = "PlotPerBatchLossHistory",
```

```

initialize = function(file = "training_loss.pdf") {
  private$outfile <- file
},

on_train_begin = function(logs = NULL) {
  private$plots_dir <- tempfile()
  dir.create(private$plots_dir)
  private$per_batch_losses <-
    fastmap::faststack(init = self$params$steps)
},

on_epoch_begin = function(epoch, logs = NULL) {
  private$per_batch_losses$reset()
},

on_batch_end = function(batch, logs = NULL) {
  private$per_batch_losses$push(logs$loss)
},

on_epoch_end = function(epoch, logs = NULL) {
  losses <- as.numeric(private$per_batch_losses$as_list())

  filename <- sprintf("epoch_%04i.pdf", epoch)
  filepath <- file.path(private$plots_dir, filename)

  pdf(filepath, width = 7, height = 5)
  on.exit(dev.off())

  plot(losses, type = "o",
        ylim = c(0, max(losses)),
        panel.first = grid(),
        main = sprintf("Training Loss for Each Batch\n(Epoch %i)", epoch),
        xlab = "Batch", ylab = "Loss")
},

on_train_end = function(logs) {
  private$per_batch_losses <- NULL
  plots <- sort(list.files(private$plots_dir, full.names = TRUE))
  qpdf::pdf_combine(plots, private$outfile)
  unlink(private$plots_dir, recursive = TRUE)
}
)

model <- get_mnist_model()
model %>% compile(optimizer = "rmsprop",
                 loss = "sparse_categorical_crossentropy",
                 metrics = "accuracy")
model %>% fit(train_images, train_labels,
            epochs = 10,
            callbacks = list(callback_plot_per_batch_loss_history()),
            validation_data = list(val_images, val_labels))

```

Мы получаем график наподобие изображенного на рис. 7.5.

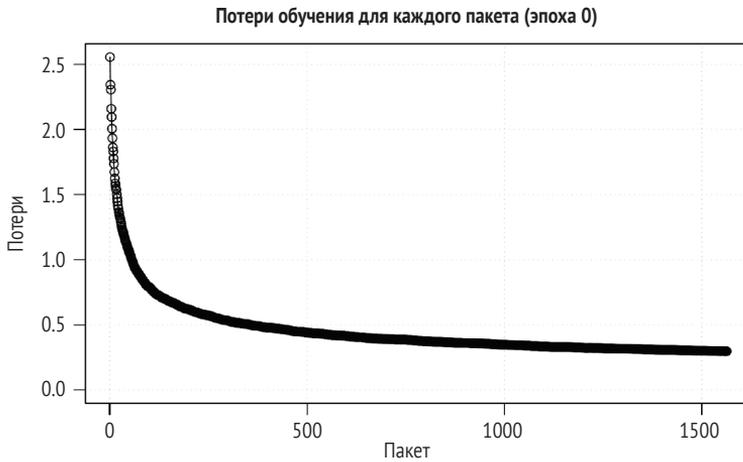


Рис. 7.5 Результат нашего пользовательского обратного вызова построения истории

### Дополнение объектов R с помощью `fastmap::faststack()`

Дополнение векторов R с помощью `c()` или `[[<-` обычно происходит медленно, и его лучше избегать. В этом примере мы используем `fastmap::faststack()`, чтобы более эффективно собирать данные о потерях для каждого пакета.

### `private` и `self` в методах пользовательских классов

Во всех предыдущих примерах для отслеживания свойств экземпляра мы использовали `self`, но в этом примере обратного вызова мы использовали `private`. Какая разница? Любое свойство, такое как `self$foo`, также доступно непосредственно из экземпляра класса в `instance$foo`. Однако свойства `private` доступны только внутри методов класса.

Еще одно важное отличие заключается в том, что Keras автоматически преобразует все, что связано с `self`, в родной формат Keras. Это помогает Keras автоматически находить, например, все переменные `tf.Variables`, связанные с пользовательским слоем. Однако это автоматическое преобразование иногда может повлиять на производительность или даже привести к сбою для определенных типов объектов R (например, `faststack()`). В свою очередь, `private` представляет собой простую среду R, которую Keras не затрагивает. Только методы класса, которые вы разрабатываете, будут напрямую взаимодействовать со свойствами `private`.

## 7.3.4 Мониторинг и визуализация с помощью TensorBoard

Чтобы провести хорошее исследование или разработать качественные модели, вам нужны подробные и регулярные сведения о том,

что происходит внутри ваших моделей во время экспериментов. В этом и заключается смысл проведения экспериментов: получить информацию о том, насколько хорошо работает модель, – как можно больше информации. Достижение прогресса – это повторяющийся процесс, замкнутый цикл: вы начинаете с идеи и выражаете ее в виде эксперимента, пытаетесь подтвердить или опровергнуть.

Вы запускаете эксперимент и обрабатываете полученную информацию. Это вдохновляет вас на следующую идею. Чем больше итераций этого цикла вы сможете выполнить, тем более совершенными и глубокими станут ваши идеи. Keras помогает вам перейти от идеи к эксперименту за минимально возможное время, а быстрые графические процессоры помогают максимально быстро перейти от эксперимента к результату. А как насчет обработки результатов? Вот тут-то и вступает в игру TensorBoard (рис. 7.6).



Рис. 7.6 Цикл прогресса

TensorBoard (<http://www.tensorflow.org/tensorboard>) – это браузерное приложение, которое можно запускать локально. Это лучший способ отслеживать все, что происходит внутри вашей модели во время обучения. С TensorBoard вы можете:

- визуально контролировать метрики во время обучения;
- визуализировать архитектуру вашей модели;
- визуализировать гистограммы активаций и градиентов;
- исследовать представления в 3D.

Если вам доступно больше информации, чем просто окончательные потери модели, вы можете получить более четкое представление о том, что модель делает и чего не делает, и намного быстрее добиться прогресса. Самый простой способ использовать TensorBoard с моделью Keras и методом `fit()` – применить обратный вызов `callback_tensorboard()`. В самом простом варианте просто укажите место для записи журнала обратного вызова, и все готово:

```
model <- get_mnist_model()
model %>% compile(optimizer = "rmsprop",
                 loss = "sparse_categorical_crossentropy",
                 metrics = "accuracy")
model %>% fit(train_images, train_labels,
            epochs = 10,
```

Путь к вашему журналу

```
validation_data = list(val_images, val_labels),
callbacks = callback_tensorboard(log_dir = "logs/"))
```

Во время работы модели обратный вызов будет записывать журналы в указанном месте. Затем вы можете просмотреть журналы, вызвав `tensorboard()`; эта команда запустит браузер с работающим приложением TensorBoard:

```
tensorboard(log_dir = "logs/") ← Запускает браузер с TensorBoard
```

В панели TensorBoard вы можете просматривать актуальные графики показателей обучения и оценки (рис. 7.7).

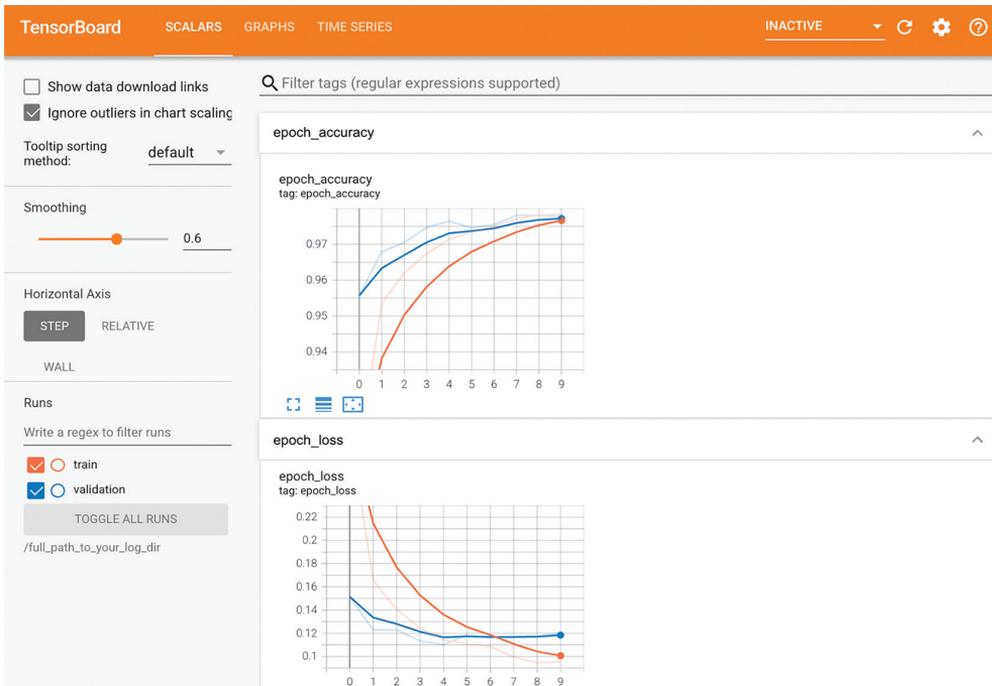


Рис. 7.7 TensorBoard можно использовать для удобного мониторинга показателей обучения и оценки

## 7.4 Разработка собственных циклов обучения и оценки

Стандартный рабочий процесс обучения `fit()` обеспечивает хороший баланс между простотой использования и гибкостью. Вы будете использовать его в большинстве случаев. Однако он не способен удовлетворить все потребности исследователей глубокого обучения,

даже если применить пользовательские метрики, функции потерь и обратные вызовы.

Дело в том, что встроенный рабочий процесс `fit()` ориентирован исключительно на обучение с учителем. Это сценарий, в котором есть известные цели (также называемые метками, или аннотациями), сопоставленные с вашими входными данными, и где вы вычисляете функцию потерь как разницу между целями и прогнозами. Однако не все формы машинного обучения подпадают под эту категорию. Существуют и другие схемы, в которых отсутствуют явные цели, такие как *генеративное обучение* (которое мы обсудим в главе 12), *самообучение* (когда цели извлекаются из входных данных) и *обучение с подкреплением* (когда обучение зависит от «вознаграждения», что очень похоже на дрессировку собаки). Даже если вы предпочитаете исследовать только обучение с подкреплением, вам могут понадобиться решения, требующие гибкости на низком уровне.

Всякий раз, когда вам недостаточно встроенного метода `fit()`, следует разработать собственную логику обучения. Вы уже видели простые примеры низкоуровневых циклов обучения в главах 2 и 3. Напомню, что содержание типичного цикла обучения выглядит следующим образом:

- 1 запуск прямого прохода (вычисление выходных данных модели) внутри градиентной ленты, чтобы получить значение потерь для текущего пакета данных;
- 2 получение градиентов потерь относительно весов модели;
- 3 обновление весов модели, чтобы снизить значение потерь в текущем пакете данных.

Эти шаги повторяются для необходимого количества пакетов. По сути, именно это и делает `fit()` за кадром. Далее вы научитесь разрабатывать свою реализацию `fit()` с нуля и получите знания, необходимые для построения любого алгоритма обучения, который вы можете придумать.

### 7.4.1 Обучение или логический вывод

В примерах низкоуровневого цикла обучения, которые вы видели до сих пор, шаг 1 (прямой проход) выполнялся с помощью `predictions <- model(inputs)`, а шаг 2 (извлечение градиентов, вычисленных градиентной лентой) выполнялся с помощью `gradients <- tape$gradient(loss, model$weights)`. В общем случае нужно учитывать две тонкости.

Некоторые слои Keras, такие как `layer_dropout()`, ведут себя по-разному во время *обучения* и во время *вывода* (когда вы используете их для создания прогнозов). Такие слои предоставляют логический аргумент `training` в своем методе `call()`. Вызов `dropout(inputs, training = TRUE)` приведет к удалению некоторых записей активации, тогда как вызов `dropout(inputs, training = FALSE)` ничего не

делает. Кроме того, функциональные и последовательные модели также предоставляют этот аргумент `training` в своих методах `call()`. Не забудьте передать аргумент `training = TRUE`, когда вы вызываете модель Keras во время прямого прохода! Таким образом, наш прямой проход имеет вид `predictions <- model(inputs, training = TRUE)`.

Кроме того, обратите внимание, что когда вы получаете градиенты весов вашей модели, вы должны использовать не `tape$gradients(loss, model$weights)`, а `tape$gradients(loss, model$trainable_weights)`. На самом деле слои и модели имеют два типа весов:

- *обучаемые веса* – они предназначены для обновления с помощью обратного распространения, чтобы свести к минимуму потери модели;
- *необучаемые веса* – они предназначены для обновления во время прямого прохода слоями, которым они принадлежат. Например, если вы хотите, чтобы пользовательский слой хранил счетчик того, сколько пакетов он обработал до сих пор, эта информация будет храниться в необучаемом весе, и после каждого пакета ваш слой будет увеличивать счетчик на единицу.

Среди встроенных слоев Keras единственным слоем, имеющим необучаемые веса, является `layer_batch_normalization()`, который мы обсудим в главе 9. Слою пакетной нормализации нужны необучаемые веса для отслеживания информации о среднем и стандартном отклонении данных, которые проходят через него. Это чтобы выполнить онлайн-аппроксимацию нормализации признаков (концепция, о которой вы узнали в главе 6). Принимая во внимание эти две детали, этап обучения с учителем в конечном итоге выглядит следующим образом:

```
library(tensorflow)

train_step <- function(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_lists(gradients, model$trainable_weights))
}
```

Вы узнали про `zip_lists()`  
в главе 2

## 7.4.2 Использование метрик на низком уровне

В низкоуровневом цикле обучения вам, вероятно, придется использовать метрики Keras (как пользовательские, так и встроенные). Вы уже знаете, как работает API метрик: просто вызовите `update_state(y_true, y_pred)` для каждой партии целей и прогнозов, а затем используйте `result()` для запроса текущего значения метрики:

```
metric <- metric_sparse_categorical_accuracy()
targets <- c(0, 1, 2)
predictions <- rbind(c(1, 0, 0),
                    c(0, 1, 0),
                    c(0, 0, 1))
metric$update_state(targets, predictions)
current_result <- metric$result()
sprintf("result: %.2f", as.array(current_result))
```

as.array конвертирует  
тензор в массив R

```
[1] "result: 1.00"
```

Если понадобится отслеживать среднее значение скаляра, например потери модели, вы можете сделать это с помощью `metric_mean()`:

```
values <- c(0, 1, 2, 3, 4)
mean_tracker <- metric_mean()
for (value in values)
  mean_tracker$update_state(value)
sprintf("Mean of values: %.2f", as.array(mean_tracker$result()))
```

```
[1] "Mean of values: 2.00"
```

Не забудьте использовать `metric$reset_state()`, если вы хотите сбросить текущие результаты (в начале эпохи обучения или в начале оценки).

### 7.4.3 Полный цикл обучения и оценки

В этом разделе мы объединим прямой проход, обратный проход и отслеживание метрик в функцию шага обучения наподобие `fit()`, которая принимает пакет данных и целей и возвращает журналы для отображения на индикаторе выполнения `fit()`.

#### Листинг 7.21 Разработка цикла обучения: функция шага обучения

```
model <- get_mnist_model()
loss_fn <- loss_sparse_categorical_crossentropy()
optimizer <- optimizer_rmsprop()
metrics <- list(metric_sparse_categorical_accuracy())
loss_tracking_metric <- metric_mean()

train_step <- function(inputs, targets) {
  with(tf$GradientTape() %as% tape, {
    predictions <- model(inputs, training = TRUE)
    loss <- loss_fn(targets, predictions)
  })
  gradients <- tape$gradient(loss,
                             model$trainable_weights)
```

Подготовка  
функции потерь

Подготовка оптимизатора

Подготовка  
списка метрик  
для отображения

Подготовка  
трекера  
`metric_mean()`  
для  
отслеживания  
средней  
потери

Запуск прямого прохода.

Учтите, что надо передать `training = TRUE`

Запуск обратного прохода.

Учтите, что надо использовать `model$trainable_weights`

```
optimizer$apply_gradients(zip_lists(gradients,
                                    model$strainable_weights))

logs <- list()
for (metric in metrics) { ←———— Отслеживаем метрики
  metric$update_state(targets, predictions)
  logs[[metric$name]] <- metric$result()
}

loss_tracking_metric$update_state(loss) ←———— Отслеживаем среднее потерь
logs$loss <- loss_tracking_metric$result()
logs ←———— Возвращаем текущие значения метрик и потерь
}
```

Нам нужно будет сбросить состояние наших метрик в начале каждой эпохи и перед выполнением оценки. В листинге 7.22 приведена вспомогательная функция для этого.

#### Листинг 7.22 Разработка цикла обучения: сброс метрик

```
reset_metrics <- function() {
  for (metric in metrics)
    metric$reset_state()
  loss_tracking_metric$reset_state()
}
```

Теперь мы можем собрать полный цикл обучения. Обратите внимание, что мы используем объект набора данных TensorFlow из пакета `tfdatasets`, чтобы превратить наши данные массива R в итератор, который перебирает данные пакетами размером 32. Механика идентична итератору набора данных, который мы реализовали в главе 2, отличаются только имена. Мы создаем экземпляр набора данных TensorFlow из наших массивов R с помощью `tensor_slices_dataset()`, преобразуем его в итератор с помощью `as_iterator()`, а затем циклически применяем `iter_next()` к итератору, чтобы получить следующий пакет. Единственное отличие заключается в том, что `iter_next()` возвращает объекты Tensor, а не массивы R. Мы расскажем больше о `tfdatasets` в главе 8.

#### Листинг 7.23 Разработка цикла обучения: непосредственно цикл

```
library(tfdatasets)
training_dataset <-
  list(inputs = train_images, targets = train_labels) %>%
  tensor_slices_dataset() %>%
  dataset_batch(32)

epochs <- 3
for (epoch in seq(epochs)) {
  reset_metrics()
  training_dataset_iterator <- as_iterator(training_dataset)
  repeat {
```

```

batch <- iter_next(training_dataset_iterator)
if (is.null(batch)) ←———— iterator исчерпан
  break
logs <- train_step(batch$inputs, batch$targets)
}

writelines(c(
  sprintf("Results at the end of epoch %s", epoch),
  sprintf("...%s: %.4f", names(logs), sapply(logs, as.numeric))
))
}

```

```

Results at the end of epoch 1
...sparse_categorical_accuracy: 0.9156
...loss: 0.2687
Results at the end of epoch 2
...sparse_categorical_accuracy: 0.9539
...loss: 0.1659
Results at the end of epoch 3
...sparse_categorical_accuracy: 0.9630
...loss: 0.1371

```

Дальше следует цикл оценки: простой цикл `for` многократно вызывает функцию `test_step()`, обрабатывающую один пакет данных. Функция `test_step()` – это просто подмножество кода функции `train_step()`. В ней отсутствует код, связанный с обновлением весов модели, то есть все, что связано с `GradientTape()` и оптимизатором.

#### Листинг 7.24 Разработка пошагового цикла оценки

```

test_step <- function(inputs, targets) {
  predictions <- model(inputs, training = FALSE) ←———— Обратите внимание,
  loss <- loss_fn(targets, predictions)           мы передаем training
                                                  = FALSE
  logs <- list()
  for (metric in metrics) {
    metric$update_state(targets, predictions)
    logs[[paste0("val_", metric$name)]] <- metric$result()
  }

  loss_tracking_metric$update_state(loss)
  logs[["val_loss"]] <- loss_tracking_metric$result()
  logs
}

val_dataset <- list(val_images, val_labels) %>%
  tensor_slices_dataset() %>%
  dataset_batch(32)

reset_metrics()

val_dataset_iterator <- as_iterator(val_dataset)
repeat {

```

```

batch <- iter_next(val_dataset_iterator)
if(is.null(batch)) break
c(inputs_batch, targets_batch) %<-% batch
logs <- test_step(inputs_batch, targets_batch)
}

writelines(c(
  "Evaluation results:",
  sprintf("...%s: %.4f", names(logs), sapply(logs, as.numeric))
))

```

iter\_next() возвращает NULL, как только итератор пакета набора данных исчерпан

```

Evaluation results:
...val_sparse_categorical_accuracy: 0.9461
...val_loss: 0.1871

```

Поздравляю – вы только что заново реализовали методы `fit()` и `evaluate()`! Точнее, почти реализовали: `fit()` и `evaluate()` поддерживают гораздо больше функций, включая крупномасштабные распределенные вычисления, а это не так просто. Они также предоставляют несколько ключевых способов оптимизации производительности. Давайте рассмотрим один из таких способов – компиляцию функции TensorFlow.

## 7.4.4 Увеличьте быстродействие с помощью `tf_function()`

Возможно, вы заметили, что ваши пользовательские циклы работают значительно медленнее, чем встроенные функции `fit()` и `evaluate()`, несмотря на то что реализуют, по существу, ту же логику. Это связано с тем, что по умолчанию код TensorFlow выполняется построчно, т. е. *непосредственно*, как и обычный код R с использованием массивов R. Непосредственное выполнение упрощает отладку кода, но далеко не оптимально с точки зрения быстродействия.

Более эффективно было бы *скомпилировать* ваш код TensorFlow в *вычислительный граф*, который допускает глобальную оптимизацию, в отличие от построчного выполнения кода. Синтаксис оптимизации очень прост: достаточно вызвать `tf_function()` для любой функции, которую вы хотите скомпилировать перед ее выполнением, как показано в листинге 7.25.

**Листинг 7.25** Использование `tf_function()` с функцией оценки

```

Передаем ранее определенный test_step
      в tf_function()
tf_test_step <- tf_function(test_step)

val_dataset_iterator <- as_iterator(val_dataset)
reset_metrics()

while(!is.null(iter_next(val_dataset_iterator) -> batch)) {
  c(inputs_batch, targets_batch) %<-% batch
}

```

Повторно используем объект Dataset, определенный в предыдущем примере, но создаем новый итератор

```

logs <- tf_test_step(inputs_batch, targets_batch)
}
writeLines(c(
  "Evaluation results:",
  sprintf("...%s: %.4f", names(logs), sapply(logs, as.numeric))
))

```

← На этот раз используем  
скомпилированную функцию `test_step`

```

Evaluation results:
...val_sparse_categorical_accuracy: 0.5190
...val_loss: 1.6764

```

На моем компьютере функция оценки без оптимизации выполняется за 2,4 секунды, а с оптимизацией всего за 0,6 секунды. Разница очевидна!

Увеличение быстродействия еще заметнее, когда цикл итерации TF Dataset также скомпилирован как графовая операция. Вы можете использовать `tf_function()` для компиляции полного цикла оценки следующим образом:

```

my_evaluate <- tf_function(function(model, dataset) {
  reset_metrics()

  for (batch in dataset) {
    c(inputs_batch, targets_batch) %<-% batch
    logs <- test_step(inputs_batch, targets_batch)
  }
  logs
})

system.time(my_evaluate(model, val_dataset))["elapsed"]

elapsed
0.283

```

И снова время выполнения сокращается более чем вдвое!

Помните, что пока вы отлаживаете свой код, лучше выполнять его построчно, без использования `tf_function()`. Так легче отслеживать ошибки. Как только ваш код заработает и вы захотите сделать его быстрым, примените декоратор `tf_function()` к этапу обучения и этапу оценки или к любой другой функции, существенно влияющей на быстродействие.

## 7.4.5 Использование `fit()` с пользовательским циклом обучения

В предыдущих разделах мы разработали собственный цикл обучения полностью с нуля. Это обеспечивает наибольшую гибкость, но в конечном итоге вы пишете много кода и одновременно теряете многие удобные возможности `fit()`, такие как обратные вызовы или встроенную поддержку распределенного обучения.

Что делать, если вам нужно реализовать собственный алгоритм обучения, но вы все равно хотите использовать мощь встроенной логики обучения Keras? На самом деле существует золотая середина между стандартным методом `fit()` и пользовательским циклом обучения, написанным с нуля, – вы можете предоставить пользовательскую функцию шага обучения и позволить фреймворку сделать все остальное.

Для этого нужно переопределить метод `train_step()` класса `Model`. Это функция, вызываемая методом `fit()` для каждого пакета данных. После этого вы сможете вызывать `fit()` как обычно, но за кулисами модели будет выполняться ваш собственный алгоритм обучения. Вот простой пример:

- создаем новый класс, который является подклассом `Model`, вызывая `new_model_class()`;
- переопределяем метод `train_step(data)`. Его содержимое почти идентично тому, что мы использовали в предыдущем разделе. Он возвращает именованный список, сопоставляющий имена метрик (включая потери) с их текущими значениями;
- реализуем активное свойство `metrics`, которое отслеживает экземпляры класса `Metric` модели. Это позволяет модели автоматически вызывать `reset_state()` для метрик модели в начале каждой эпохи и в начале вызова `evaluate()`, поэтому вам не нужно делать это вручную:

```

loss_fn <- loss_sparse_categorical_crossentropy()
loss_tracker <- metric_mean(name = "loss")

CustomModel <- new_model_class(
  classname = "CustomModel",
  train_step = function(data) {
    c(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_lists(gradients, model$trainable_weights))

    loss_tracker$update_state(loss)
    list(loss = loss_tracker$result())
  },
  metrics = mark_active(function() list(loss_tracker))
)

```

Этот объект метрики будет использоваться для отслеживания средних потерь на партию во время обучения и оценки

Мы переопределяем метод `train_step`

Мы используем `self(inputs, training = TRUE)` вместо `model(inputs, training = TRUE)`, потому что наша модель – это сам класс

Мы обновляем метрику отслеживания потерь, которая отслеживает среднее значение потерь

Мы возвращаем средние потери на данный момент, запрашивая метрику отслеживания потерь

Любая метрика, которую вы хотите сбросить между эпохами, должна быть указана здесь

Теперь мы можем создать экземпляр нашей пользовательской модели, скомпилировать ее (мы передаем только оптимизатор, потому что потери уже определены вне модели) и обучить ее, используя `fit()`, как обычно:

```
inputs <- layer_input(shape = c(28 * 28))
features <- inputs %>%
  layer_dense(512, activation = "relu") %>%
  layer_dropout(0.5)
outputs <- features %>%
  layer_dense(10, activation = "softmax")

model <- CustomModel(inputs = inputs, outputs = outputs)

model %>% compile(optimizer = optimizer_rmsprop())
model %>% fit(train_images, train_labels, epochs = 3)
```

Поскольку мы не предоставили метод `initialize()`, используется та же сигнатура, что и `keras_model()`: входные данные, выходные данные и, возможно, имя

Здесь есть пара замечаний:

- этот подход не мешает вам создавать модели с помощью Functional API. Вы можете сделать это независимо от того, какие модели вы создаете: последовательные, функциональные или подклассовые;
- вам не нужно вызывать `tf_function()`, когда вы переопределяете `train_step`, – фреймворк сделает это за вас.

А как насчет метрик и настройки функции потерь через `compile()`? После вызова `compile()` вы получаете доступ к следующему:

- `self$compiled_loss` – функция потери, которую вы передали в `compile()`;
- `self$compiled_metrics` – обертка для списка переданных вами метрик, которая позволяет вызывать `self$compiled_metrics$update_state()` для одновременного обновления всех ваших метрик;
- `self$metrics` – фактический список метрик, которые вы передали в `compile()`. Он также включает метрику, которая отслеживает потери, аналогично тому, что мы делали вручную с нашей метрикой `loss_tracking_metric` ранее.

Таким образом, мы можем написать:

```
CustomModel <- new_model_class(
  classname = "CustomModel",

  train_step = function(data) {
    c(inputs, targets) %<-% data
    with(tf$GradientTape() %as% tape, {
      predictions <- self(inputs, training = TRUE)
      loss <- self$compiled_loss(targets, predictions)
    })
    gradients <- tape$gradient(loss, model$trainable_weights)
```

Вычисление потерь через `self$compiled_loss`

```

optimizer$apply_gradients(zip_lists(gradients, model$trainable_weights))
self$compiled_metrics$update_state(
  targets, predictions)
results <- list()
for(metric in self$metrics)
  results[[metric$name]] <- metric$result()
results
}
)

```

Обновление метрик модели  
через self\$compiled\_metrics

Возвращение именованного  
списка значений метрик

Давайте попробуем обучить модель:

```

inputs <- layer_input(shape = c(28 * 28))
features <- inputs %>%
  layer_dense(512, activation = "relu") %>%
  layer_dropout(0.5)

outputs <- features %>% layer_dense(10, activation = "softmax")
model <- CustomModel(inputs = inputs, outputs = outputs)

model %>% compile(optimizer = optimizer_rmsprop(),
  loss = loss_sparse_categorical_crossentropy(),
  metrics = metric_sparse_categorical_accuracy())
model %>% fit(train_images, train_labels, epochs = 3)

```

В этой главе на вас обрушилась гора информации, но теперь вы знаете достаточно, чтобы использовать Keras практически в любой ситуации.

## Краткие итоги главы

- Keras предлагает широкий выбор различных рабочих процессов, основанный на принципе *постепенного раскрытия сложности*. Все они хорошо работают в различном сочетании.
- Вы можете создавать модели через Sequential API с помощью `keras_model_sequential()`, через Functional API с помощью `keras_model()` или путем создания подкласса класса `Model` с помощью `new_model_class()`. В основном вы будете использовать Functional API.
- Самый простой способ обучить и оценить модель – использовать методы по умолчанию `fit()` и `evaluate()`.
- Обратные вызовы Keras обеспечивают простой способ мониторинга моделей во время вашего вызова `fit()` и автоматически предпринимают действия в зависимости от состояния модели.
- Вы можете полностью контролировать все операции `fit()`, переопределив метод `train_step()`.
- Помимо метода `fit()`, вы также можете создавать собственные циклы обучения. Это полезно для исследователей, разрабатывающих совершенно новые алгоритмы обучения.

# Глубокое обучение в компьютерном зрении

---

## **Эта глава охватывает следующие темы:**

- суть сверточных нейронных сетей;
- расширение обучающего набора данных для ослабления эффекта переобучения;
- использование предварительно обученной сверточной нейронной сети для извлечения признаков;
- дообучение предварительно обученной сверточной нейронной сети.

Компьютерное зрение – самая первая и самая значительная история успеха глубокого обучения. Каждый день вы взаимодействуете с моделями глубокого зрения – через Google Photos, поиск изображений Google, YouTube, видеофильтры в приложениях камеры, приложения оптического распознавания текстов и многое другое. Эти модели также лежат в основе передовых исследований в области автономного вождения, робототехники, медицинской диагностики с помощью ИИ, автономных кассовых систем и даже автономного земледелия.

Компьютерное зрение стало той прикладной областью, которая привела к первому прорыву в развитии глубокого обучения в период с 2011 по 2015 год. Одна из архитектур глубокого обучения, так называемая *сверточная нейронная сеть*, примерно в то же время

начала давать удивительно хорошие результаты на соревнованиях по классификации изображений (конкурс по распознаванию китайских иероглифов ICDAR 2011 и конкурс по распознаванию немецких дорожных знаков IJCNN 2011), а затем, что более важно, осенью 2012 года группа Хинтона с большим отрывом выиграла конкурс визуального распознавания цифр ImageNet. После этого многообещающие результаты начали появляться и в других задачах компьютерного зрения.

Примечательно, что первых успехов (пусть и выдающихся) было недостаточно, чтобы глубокое обучение нашло массовое применение, – на это ушло несколько лет. Сообщество исследователей компьютерного зрения потратило много сил и времени на методы, отличные от нейронных сетей, и не было готово отказаться от старых идей только потому, что по соседству появился амбициозный новичок. В 2013 и 2014 годах глубокое обучение все еще наталкивалось на глубокий скептицизм со стороны опытных исследователей компьютерного зрения. Только в 2016 году оно заняло доминирующее положение. Я помню, как в феврале 2014 года уговаривал своего бывшего профессора заняться глубоким обучением и утверждал, что эту технологию ждет великое будущее. «Ну, не знаю... возможно, это просто модная забава», – ответил он. К 2016 году вся его лаборатория занималась глубоким обучением. Идею, время которой пришло, не остановить.

Эта глава знакомит со сверточными нейронными сетями – разновидностью моделей глубокого обучения, почти повсеместно используемых в приложениях распознавания образов. Здесь вы научитесь применять сверточные нейронные сети для решения задач классификации изображений, в частности задач с небольшими наборами обучающих данных, которые наиболее распространены, 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, в которой сеть используется для классификации изображений цифр из набора MNIST. Эту задачу мы решили в главе 2, используя полносвязную сеть (ее точность на контрольных данных составила 97,8 %). Несмотря на простоту сверточной нейронной сети, ее точность будет значительно выше полносвязной модели из главы 2.

В листинге 8.1 показано, как выглядит простая сверточная нейронная сеть. Это стек слоев `layer_conv_2d` и `layer_max_pooling_2d`. Как она действует, мы расскажем чуть позже.

**Листинг 8.1** Создание небольшой сверточной нейронной сети

```
inputs <- layer_input(shape = c(28, 28, 1))

outputs <- inputs %>%
  layer_conv_2d(filters = 32, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 64, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 128, kernel_size = 3, activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(10, activation = "softmax")

model <- keras_model(inputs, outputs)
```

Важно отметить, что данная сеть принимает на входе тензоры с формой (`image_height`, `image_width`, `image_channels`), не включая измерение, относящееся к пакетам. 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).

**Листинг 8.2** Просмотр сводной информации о модели

```
model
```

```
Model: "model"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d (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 (`height`, `width`, `channels`). Измерения ширины и высоты сжимаются с ростом глубины сети. Количество каналов определяется первым аргументом, передаваемым в слои `layer_conv_2d()` (32, 64 или 128).

На выходе после последнего слоя Conv2D мы получаем результат формы (3, 3, 128) – карту признаков 3×3 из 128 каналов. Следующ-

щим шагом является подача этого вывода в полносвязный классификатор, подобный тем, с которыми вы уже знакомы: стек слоев Dense. Эти классификаторы обрабатывают векторы, которые являются одномерными, тогда как текущий результат представляет собой трехмерный тензор. Чтобы устранить несовпадение форм, мы приводим трехмерные результаты к одномерным с помощью слоя Flatten перед слоями Dense. Наконец, мы выполняем классификацию по 10 классам, поэтому наш последний слой имеет 10 выходов и активацию softmax.

Теперь давайте обучим сверточную сеть на цифрах MNIST. Мы повторно используем большую часть кода из примера MNIST в главе 2. Поскольку мы выполняем 10-классовую классификацию с выводом softmax, в качестве функции потерь будем использовать категориальную кросс-энтропийную потерю, а поскольку наши метки являются целыми числами, будем использовать разреженную версию sparse\_categorical\_crossentropy.

### Листинг 8.3 Обучение сверточной нейронной сети на изображениях MNIST

```
c(c(train_images, train_labels), c(test_images, test_labels)) %<-%
  dataset_mnist()
train_images <- array_reshape(train_images, c(60000, 28, 28, 1)) / 255
test_images <- array_reshape(test_images, c(10000, 28, 28, 1)) / 255

model %>% compile(optimizer = "rmsprop",
                  loss = "sparse_categorical_crossentropy",
                  metrics = c("accuracy"))
model %>% fit(train_images, train_labels, epochs = 5, batch_size = 64)
```

Оценим модель на контрольных данных:

### Листинг 8.4 Оценивание сверточной сети

```
result <- evaluate(model, test_images, test_labels)
cat("Test accuracy:", result['accuracy'], "\n")
```

```
| Test accuracy: 0.9915
```

Полносвязная сеть из главы 2 показала точность 97,8 % на контрольных данных, а простенькая сверточная нейронная сеть показала точность 99,1 %: мы уменьшили процент ошибок на 60 % (относительно). Неплохо!

Но почему такая простая сверточная нейронная сеть работает настолько хорошо в сравнении с полносвязной моделью? Чтобы ответить на этот вопрос, нужно разобраться во всех деталях работы слоев Conv2D и MaxPooling2D.

### 8.1.1 Операция свертки

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), а 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  $3 \times 3$ .

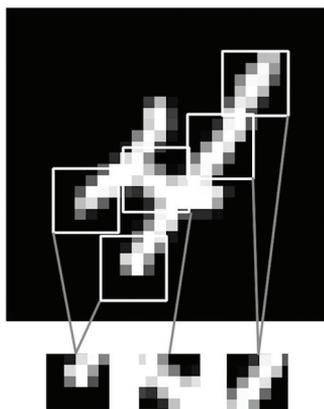


Рис. 8.1 Изображения можно разбить на локальные шаблоны, такие как края, текстуры и т. д.

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 картинке сверточная нейронная сеть сможет распознавать его повсюду: например, в левом верхнем углу. Полносвязной сети пришлось бы изучить шаблон заново, если он появляется в другом месте. Это увеличивает эффективность сверточных сетей в задачах обработки изображений (потому что видимый мир по своей сути является инвариантным в отношении переноса): таким сетям требуется меньше обучающих образцов для получения представлений, обладающих силой обобщения;
- *они могут изучать пространственные иерархии шаблонов* (рис. 8.2). Первый сверточный слой будет изучать небольшие локальные шаблоны, такие как края, второй – более крупные шаблоны, состоящие из признаков, возвращаемых первым слоем, и т. д. Это позволяет сверточным нейронным сетям эффективно изучать все более сложные и абстрактные визуальные представления (потому что видимый мир по своей сути является пространственно-иерархическим).

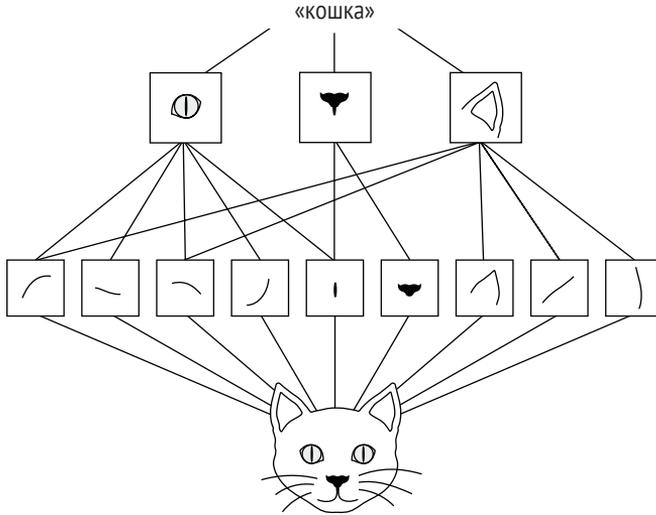


Рис. 8.2 Видимый мир формируется пространственными иерархиями видимых модулей: гиперлокальные края объединяются в локальные объекты, такие как глаза или уши, которые, в свою очередь, объединяются в понятия еще более высокого уровня, такие как «кошка»

Свертка применяется к трехмерным тензорам, называемым *картами признаков*, с двумя пространственными осями (*высота* и *ширина*), а также с осью *глубины* (или осью *каналов*). Для изображений в формате RGB размерность оси глубины равна 3, потому что имеется три канала цвета: красный (red), зеленый (green) и синий (blue). Для черно-белых изображений, как в наборе MNIST, ось глубины имеет размерность 1 (оттенки серого). Операция свертки извлекает шаблоны из своей входной карты признаков и применяет одинаковые преобразования ко всем шаблонам, производя *выходную карту признаков*. Эта выходная карта признаков также является трехмерным тензором: она имеет ширину и высоту. Ее глубина может иметь любую размерность, потому что выходная глубина является параметром слоя, и разные каналы на этой оси глубины больше не соответствуют конкретным цветам, как во входных данных в формате RGB; по сути, они являются *фильтрами*. Фильтры представляют конкретные аспекты входных данных: на верхнем уровне, например, фильтр может соответствовать понятию «присутствие лица на входе».

В примере MNIST первый сверточный слой принимает карту признаков с размером (28, 28, 1) и выводит карту признаков с размером (26, 26, 32): он вычисляет 32 фильтра по входным данным. Каждый из этих 32 выходных каналов содержит сетку 26×26 значений – *карту отклика* фильтра на входных данных, определяющую ответ этого шаблона фильтра для разных участков входных данных (рис. 8.3).

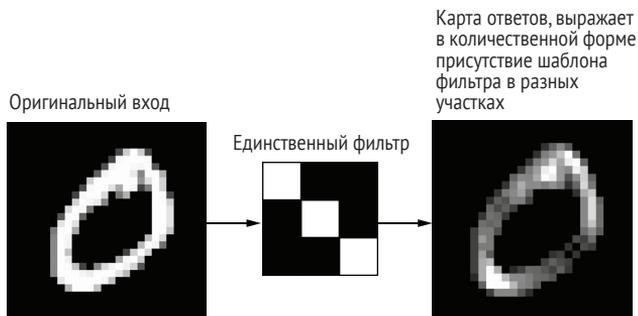


Рис. 8.3 Понятие карты отклика: двумерная карта наличия шаблона в разных участках входных данных

Вот что означает термин *карта признаков*: каждое измерение на оси глубины – это *признак* (или *фильтр*), а двумерный тензор `output[ , , n]` – это двумерная пространственная карта отклика этого 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 вариантом;
- *глубиной выходной карты признаков* – количеством фильтров, вычисляемых сверткой. В данном примере свертка начинается с глубины 32 и заканчивается глубиной 64.

В `layer_conv_2d()` эти параметры являются первыми аргументами, передаваемыми слою (после `inputs`): `inputs %>% layer_conv_2d(output_depth, c(window_height, window_width))`.

Свертка работает методом *скользящего окна*: она двигает окно с размером  $3 \times 3$  или  $5 \times 5$  по трехмерной входной карте признаков, останавливается в каждой возможной позиции и извлекает трехмерный шаблон окружающих признаков с формой (`window_height`, `window_width`, `input_depth`). 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 (`output_depth`). Все эти векторы затем собираются в трехмерную выходную карту с формой (высота, ширина, выходная глубина). Все эти векторы (по одному на шаблон) затем пространственно пересобираются в выходную трехмерную карту формы (`height`, `width`, `output_depth`). Каждое пространственное положение на выходной карте объектов соответствует одному и тому же местоположению на входной карте объектов (например, правый нижний угол входных данных содержит информацию о правом нижнем углу входных данных). Например, для окна  $3 \times 3$  вектор `output[i, j, ]` соответствует трехмерному шаблону `input[(i-1):(i+1), (j-1):(j+1), ]`. Полный процесс изображен на рис. 8.4.

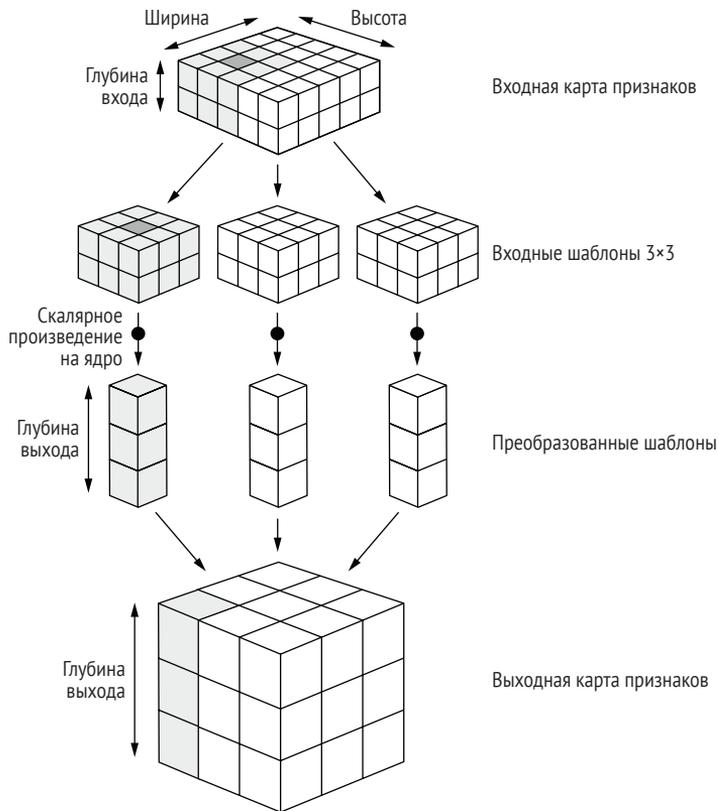


Рис. 8.4 Принцип действия свертки

Обратите внимание, что выходные ширина и высота могут отличаться от входных. На то есть две причины:

- эффекты границ, которые могут устраняться дополнением входной карты признаков;
- использование *шага свертки* (*страйд*, *stride*), определение которого приводится чуть ниже.

Рассмотрим подробнее эти понятия.

### ЭФФЕКТЫ ГРАНИЦ И ДОПОЛНЕНИЕ

Рассмотрим карту признаков  $5 \times 5$  (всего 25 клеток). Существует всего 9 клеток, в которых может находиться центр окна  $3 \times 3$ , образующих сетку  $3 \times 3$  (рис. 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  $26 \times 26$ .

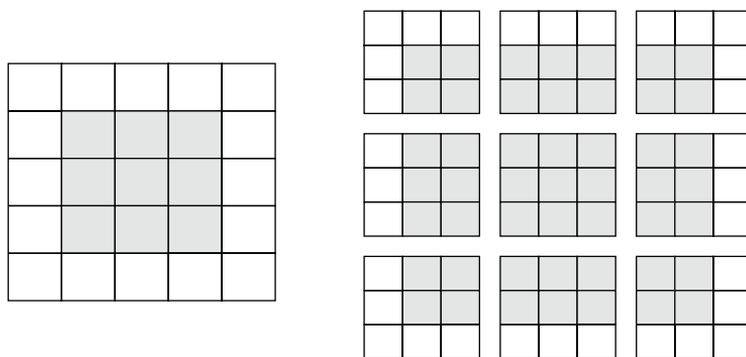


Рис. 8.5 Допустимые местоположения шаблонов 3×3 во входной карте признаков 5×5

Чтобы получить выходную карту признаков с теми же пространственными размерами, что и входная карта, можно использовать *дополнение* (padding). Дополнение заключается в добавлении соответствующего количества строк и столбцов с каждой стороны входной карты признаков, чтобы можно было поместить центр окна свертки в каждую входную клетку. Для окна 3×3 нужно добавить один столбец справа, один столбец слева, одну строку сверху и одну строку снизу. Для окна 5×5 нужно добавить две строки (рис. 8.6).

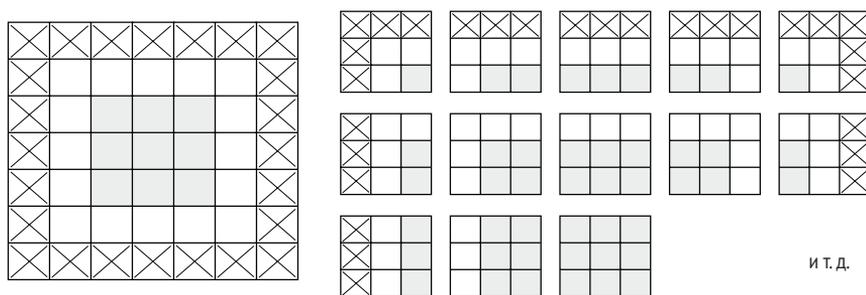


Рис. 8.6 Дополнение входной карты признаков 5×5, чтобы получить 25 шаблонов 3×3

При использовании слоев `layer_conv_2d` дополнение настраивается с помощью аргумента `padding`, который принимает два значения: "valid", означающее отсутствие дополнения (будут использоваться только допустимые местоположения окна), и "same", означающее «дополнить так, чтобы выходная карта признаков имела ту же ширину и высоту, что и входная». По умолчанию аргумент `padding` получает значение "valid".

## ПОНЯТИЕ ШАГА 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 пропусками* (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).

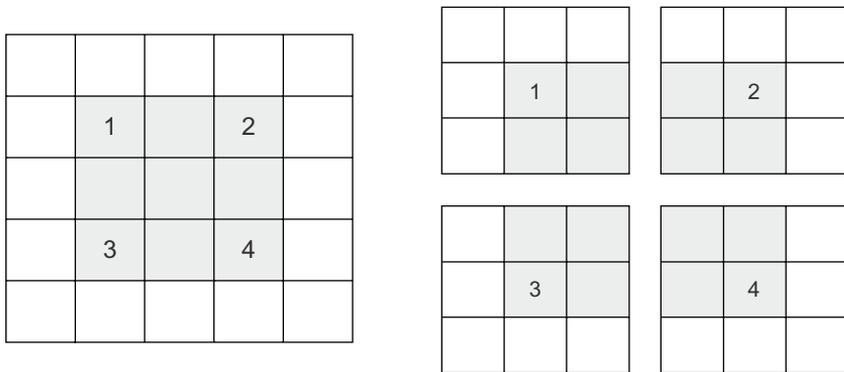


Рис. 8.7 Шаблоны  $3 \times 3$  свертки с шагом  $2 \times 2$

Использование шага 2 означает уменьшение ширины и высоты карты признаков за счет уменьшения разрешения в 2 раза (в дополнение к любым изменениям, вызванным эффектами границ). Свертки с пробелами редко используются на практике, хотя могут пригодиться в моделях некоторых типов, как вы увидите в следующей главе.

Для уменьшения разрешения карты признаков в классификаторах 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 уровня `layer_max_pooling_2d()`. Например, перед первым уровнем `layer_max_pooling_2d()` карта признаков имела размер  $26 \times 26$ , но операция выбора максимального значения из соседних уменьшила ее до раз-

мера  $13 \times 13$ . В этом заключается предназначение данной операции: агрессивное уменьшение разрешения карты признаков во многом подобно 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 `max` (выбор 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 2 раза. Свертка же, напротив, обычно выполняется с окном  $3 \times 3$  и без пропусков (шаг равен 1).

Зачем вообще снижать разрешение карты признаков? Почему бы просто не убрать уровни `layer_max_pooling_2d()` и использовать карты признаков большего размера? Рассмотрим этот вариант. Сверточная основа модели в этом случае будет выглядеть, как в листинге 8.5.

#### Листинг 8.5 Некорректно построенная сверточная сеть с отсутствующими слоями максимального значения из соседних

```
inputs <- layer_input(shape = c(28, 28, 1))
outputs <- inputs %>%
  layer_conv_2d(filters = 32, kernel_size = 3, activation = "relu") %>%
  layer_conv_2d(filters = 64, kernel_size = 3, activation = "relu") %>%
  layer_conv_2d(filters = 128, kernel_size = 3, activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(10, activation = "softmax")
model_no_max_pool <- keras_model(inputs = inputs, outputs = outputs)
```

Так выглядит сводная информация о модели:

```
model_no_max_pool
```

```
Model: "model_1"
```

```
Layer (type) Output Shape Param #
```

```
=====  
input_2 (InputLayer) [(None, 28, 28, 1)] 0  
conv2d_5 (Conv2D) (None, 26, 26, 32) 320  
conv2d_4 (Conv2D) (None, 24, 24, 64) 18496  
conv2d_3 (Conv2D) (None, 22, 22, 128) 73856  
flatten_1 (Flatten) (None, 61952) 0  
dense_1 (Dense) (None, 10) 619530  
=====
```

```
Total params: 712,202
```

Trainable params: 712,202

Non-trainable params: 0

---

Что не так в этой конфигурации? Следующие два аспекта:

- она не способствует изучению пространственной иерархии признаков. Окна  $3 \times 3$  в третьем уровне содержат только информацию, поступающую из окон  $7 \times 7$  в исходных данных. Высокоуровневые шаблоны, изученные с помощью сверточной нейронной сети, будут слишком малы в сравнении с начальными данными, чего может оказаться недостаточно для обучения классификации цифр (попробуйте распознать цифру, посмотрев на нее через окна  $7 \times 7$  пикселей!). Нам нужно, чтобы признаки после последнего сверточного уровня содержали информацию о совокупности исходных данных;
- заключительная карта признаков имеет  $22 \times 22 \times 128 = 61\,952$  коэффициентов на образец. Это очень много. Если бы вы решили сделать ее плоской, чтобы наложить сверху полносвязный слой `layer_dense` размером 10, этот слой имел бы более полумиллиона 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 уровней 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, на предыдущих сверточных уровнях можно также использовать шаг свертки. И вместо выбора максимального значения 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. Но операция выбора максимального значения обычно дает лучшие результаты, чем эти альтернативные решения. Причина в том, что признаки, как правило, кодируют пространственное присутствие некоторого шаблона или понятия в разных клетках карты признаков, поэтому максимальное присутствие признаков намного информативнее, чем среднее присутствие. Следовательно, более разумная стратегия снижения разрешения состоит в том, чтобы сначала получить плотные карты признаков (путем обычной свертки без пробелов), а затем рассмотреть максимальные значения признаков в небольших шаблонах, а не разреженные окна из входных данных (путем свертки с пропусками) или усредненные шаблоны, которые могут привести к потере информации о наличии признака.

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 – и о том, как построить небольшую сверточную нейронную сеть для решения такой простой задачи, как классификация цифр из набора MNIST. Теперь перейдем к более полезным и практичным применениям.

## 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 образцов 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 – *расширение данных* (data augmentation). С его помощью мы повысим точность классификации до 80–85 %.

В следующем разделе мы рассмотрим еще два основных приема глубокого обучения на небольших наборах данных: *извлечение признаков с использованием предварительно обученной сети* (поможет поднять точность до 97,5 %) и *дообучение предварительно обученной сети* (поможет достичь окончательной точности в 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 «достаточно большой объем данных» весьма относительно – в первую очередь в отношении размера и глубины обучаемой сети. Нельзя обучить сверточную нейронную сеть решению сложной задачи на нескольких десятках образцов, а вот нескольких сотен вполне может хватить, если модель невелика и регуляризована, а решаемая задача проста. Так как сверточные нейронные сети изучают локальные признаки, инвариантные в отношении переноса, они обладают высокой эффективностью в решении задач распознавания. Обучение сверточной нейронной сети с нуля на очень небольшом наборе изображений дает вполне неплохие результаты, несмотря на относительную нехватку данных, без необходимости конструировать признаки вручную. В данном разделе мы убедимся в этом на практике.

Более того, модели глубокого обучения по своей природе очень гибкие: можно, к примеру, обучить модель для классификации изображений или распознавания речи на очень большом наборе данных и затем использовать ее для решения самых разных задач с небольшими модификациями. В частности, в распознавании образов многие предварительно обученные модели (обычно на наборе данных ImageNet) теперь доступны всем желающим для загрузки и могут использоваться как основа для создания очень мощных моделей распознавания образов на небольших объемах данных. Именно так мы и поступим в следующем разделе. Начнем с получения данных.

### 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) (для этого вам потребуется создать учетную запись на сайте Kaggle, если у вас ее еще нет, но не волнуйтесь, процесс регистрации очень прост). Для скачивания данных вы также можете воспользоваться интерфейсом командной строки Kaggle.

#### Загрузка набора данных Kaggle

Kaggle предоставляет простой в использовании API для программной загрузки наборов данных, размещенных на сайте Kaggle. Вы можете использовать его, например, для загрузки набора данных «Dogs vs. Cats» на свой локальный компьютер. Скачать этот набор данных так же просто, как запустить одну команду в R.

Однако доступ к API открыт только для пользователей Kaggle, поэтому для запуска команды вам сначала необходимо пройти аутентификацию. Пакет `kaggle` будет искать ваши учетные данные для входа в файле JSON, расположенном по адресу `~/.kaggle/kaggle.json`. Давайте создадим этот файл.

Во-первых, вам нужно создать ключ API Kaggle и загрузить его на свой локальный компьютер. Перейдите на веб-сайт Kaggle в веб-браузере, войдите в учетную запись и перейдите на страницу **My Account** (Моя учетная запись). В настройках учетной записи вы найдете раздел API. Нажав кнопку **Create New API Token** (Создать новый токен API), вы создадите файл ключа `kaggle.json` и загрузите его на свой компьютер.

Наконец, создайте папку `~/.kaggle`. В целях безопасности вы также должны убедиться, что файл доступен для чтения лишь текущему пользователю, то есть вам. (Это применимо, только если вы используете Mac или Linux, а не Windows.)

Поскольку в следующих главах мы будем выполнять большое количество операций с файловой системой, имеет смысл установить пакет `fs` R, с которым немного удобнее работать, чем с базовыми функциями файловой системы R. Вы можете установить его из CRAN с помощью `install.packages("fs")`.

Подготовим ключ API Kaggle:

```
library(fs)
dir_create("~/kaggle")
file_move("~/Downloads/kaggle.json", "~/kaggle/")
file_chmod("~/kaggle/kaggle.json", "0600")
```

Пометим файл как доступный  
для чтения только владельцу

Установим пакет `kaggle` при помощи `pip`:

```
reticulate::py_install("kaggle", pip = TRUE)
```

Теперь вы можете скачать данные, которые мы будем использовать:

```
system('kaggle competitions download -c dogs-vs-cats')
```

При первой попытке загрузить данные вы можете получить ошибку «403 Forbidden» (доступ запрещен). Это связано с тем, что вам необходимо принять условия, связанные с набором данных, прежде чем загружать его, – вам нужно будет перейти на <http://www.kaggle.com/c/dogs-vs-cats/rules> (войдя в свою учетную запись Kaggle) и нажать кнопку **I Understand and Accept** (Я понимаю и принимаю). Достаточно сделать это только один раз.

Данные загружаются в виде сжатого файла `dogs-vs-cats.zip`. Внутри этого файла есть еще один сжатый файл `train.zip`, который представляет собой обучающие данные, которые мы будем использовать. Распакуйте `train.zip` в новый каталог `dogs-vs-cats`, используя пакет R `zip` (устанавливаемый из CRAN с помощью `install.packages("zip")`):

```
zip::unzip('dogs-vs-cats.zip', exdir = "dogs-vs-cats",
  ➡ files = "train.zip")
zip::unzip("dogs-vs-cats/train.zip", exdir = "dogs-vs-cats")
```

Этот набор содержит изображения в формате JPEG с низким разрешением. На рис. 8.8 показано несколько примеров.



Рис. 8.8 Примеры изображений из набора «Собака или кошка». Изображения имеют разные размеры, ракурсы съемки и т. д.

Неудивительно, что состязание по классификации изображений кошек и собак на сайте Kaggle в 2013 году выиграли участники, использовавшие сверточные нейронные сети. Лучшие результаты достигали точности в 95 %. В нашем примере мы приблизимся к этой точности (в следующем разделе), даже притом, что для обучения моделей будем использовать менее 10 % данных, которые были доступны участникам состязаний.

Этот набор содержит 25 000 изображений кошек и собак (по 12 500 для каждого класса) общим объемом 543 Мбайт (в сжатом виде). После загрузки и распаковки архива мы создадим новый набор, разделенный на три поднабора: обучающий набор с 1000 образцов каждого класса, проверочный набор с 500 образцами каждого класса и контрольный набор с 1000 образцов каждого класса. Зачем это делать? Потому что многие наборы данных изображений, с которыми вы столкнетесь в своей работе, содержат всего несколько тысяч образцов, а не десятки тысяч. Наличие большего количества доступных данных упростило бы задачу, поэтому рекомендуется учиться на небольшом наборе данных.

Разбитый на разделы набор данных, с которым мы будем работать, имеет следующую структуру:

```

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

```

Давайте посмотрим, что происходит после нескольких вызовов функций {fs}.

### Листинг 8.6 Копирование изображений в обучающий, проверочный и контрольный каталоги

Вспомогательная функция, которая копирует изображения кошек и собак между индексами `start_index` и `end_index` в подкаталог `new_base_dir/{subset_name}/cat` (и /dog). «subset\_name» будет «train», «validation» или «test»

```

Путь к каталогу, в котором был распакован исходный набор данных
library(fs)
original_dir <- path("dogs-vs-cats/train")
new_base_dir <- path("cats_vs_dogs_small")

Каталог, в котором мы будем хранить наш меньший набор данных
make_subset <- function(subset_name,
                        start_index, end_index) {
  for (category in c("dog", "cat")) {
    file_name <- glue::glue("{category}.{ start_index:end_index }.jpg")
    dir_create(new_base_dir / subset_name / category)
    file_copy(original_dir / file_name,
              new_base_dir / subset_name / category / file_name)
  }
}

Создайте обучающую подгруппу с первой 1000 изображений каждой категории
make_subset("train", start_index = 1, end_index = 1000)
make_subset("validation", start_index = 1001, end_index = 1500)
make_subset("test", start_index = 1501, end_index = 2500)

Создайте тестовое подмножество из следующих 1000 изображений каждой категории
Создайте подмножество проверки из следующих 500 изображений каждой категории

```

Итак, 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 Построение сети

Здесь мы реализуем ту же самую общую структуру модели, как в первом примере: сверточная нейронная сеть будет организована как стек чередующихся уровней `layer_conv_2d()` (с функцией активации `relu`) и `layer_max_pooling_2d()`.

Но так как мы имеем дело с большими изображениями и решаем более сложную задачу, мы сделаем сеть больше: она будет иметь на пару уровней `layer_conv_2d()` и `layer_max_pooling_2d()` больше.

Это увеличит ее емкость и обеспечит дополнительное снижение размеров карт признаков, чтобы они не оказались слишком большими, когда достигнут уровня `layer_flatten()`. Учитывая, что мы начнем с входов, имеющих размер  $180 \times 180$  (выбран совершенно произвольно), в конце, точно перед уровнем `layer_flatten()`, получится карта признаков размером  $7 \times 7$ .

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 (уровень `layer_dense()` с размером 1 и функцией активации `sigmoid`). Этот признак будет представлять вероятность принадлежности рассматриваемого изображения одному из двух классов.

Последнее небольшое отличие: мы начнем модель со слоя `layer_rescaling()` для масштабирования значений пикселей произвольных входных изображений (которые изначально находятся в диапазоне  $[0, 255]$ ) до диапазона  $[0, 1]$ .

### Листинг 8.7 Создание небольшой сверточной нейронной сети для классификации изображений кошек и собак

Модель ожидает RGB-изображения размером  $180 \times 180$

Приводим входные данные к диапазону  $[0, 1]$ , разделив их на 255

```
inputs <- layer_input(shape = c(180, 180, 3))
outputs <- inputs %>%
  layer_rescaling(1 / 255) %>%
  layer_conv_2d(filters = 32, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 64, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 128, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 256, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 256, kernel_size = 3, activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)
```

Посмотрим, как изменяются размеры карт признаков с каждым последующим уровнем:

```
model
```

```
Model: "model_2"
```

Layer (type)	Output Shape	Param #
=====		

```

input_3 (InputLayer)          [(None, 180, 180, 3)]      0
rescaling (Rescaling)         (None, 180, 180, 3)       0
conv2d_10 (Conv2D)            (None, 178, 178, 32)      896
max_pooling2d_5 (MaxPooling2D) (None, 89, 89, 32)        0
conv2d_9 (Conv2D)             (None, 87, 87, 64)        18496
max_pooling2d_4 (MaxPooling2D) (None, 43, 43, 64)        0
conv2d_8 (Conv2D)             (None, 41, 41, 128)       73856
max_pooling2d_3 (MaxPooling2D) (None, 20, 20, 128)       0
conv2d_7 (Conv2D)             (None, 18, 18, 256)       295168
max_pooling2d_2 (MaxPooling2D) (None, 9, 9, 256)         0
conv2d_6 (Conv2D)             (None, 7, 7, 256)         590080
flatten_2 (Flatten)           (None, 12544)              0
dense_2 (Dense)               (None, 1)                  12545
=====
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0

```

На этапе компиляции, как обычно, используем оптимизатор `RMSprop`. Так как сеть заканчивается единственным признаком, используем функцию потерь `binary_crossentropy` (напомним, что в табл. 6.1 представлен краткий справочник по использованию функций потерь в конкретных 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 сеть данные должны быть преобразованы в тензоры с вещественными числами. В настоящее время данные хранятся в виде файлов JPEG, поэтому их нужно подготовить для передачи в сеть, выполнив следующие шаги:

- 1 прочитать файлы с изображениями;
- 2 декодировать содержимое из формата JPEG в таблицы пикселей RGB;
- 3 преобразовать их в тензоры с вещественными числами;
- 4 привести все изображения к одному размеру (мы используем  $180 \times 180$ );
- 5 упаковать изображения в пакеты (мы используем пакеты из 32 изображений).

Этот порядок действий может показаться немного сложным, но, к счастью, в Keras имеются утилиты, способные выполнить его автоматически. В частности, в Keras вы найдете функцию `image_dataset_from_directory()`, которая позволяет быстро настроить конвейер

ер данных, автоматически преобразующий файлы изображений на диске в пакеты предварительно обработанных тензоров. В листинге 8.9 показано, как ею воспользоваться.

Вызов `image_dataset_from_directory(directory)` сначала выводит список подкаталогов `directory` и предполагает, что каждый из них содержит изображения, принадлежащие одному из наших классов. Затем он индексирует файлы изображений в каждом подкаталоге. Наконец, он создает и возвращает объект набора данных TensorFlow, настроенный для чтения этих файлов, их перемешивания, декодирования в тензоры, приведения к нужной размерности и упаковки их в пакеты.

### Листинг 8.9 Использование `image_dataset_from_directory` для чтения изображений

```
train_dataset <-
  image_dataset_from_directory(new_base_dir / "train",
                              image_size = c(180, 180),
                              batch_size = 32)
validation_dataset <-
  image_dataset_from_directory(new_base_dir / "validation",
                              image_size = c(180, 180),
                              batch_size = 32)
test_dataset <-
  image_dataset_from_directory(new_base_dir / "test",
                              image_size = c(180, 180),
                              batch_size = 32)
```

### Подробнее о `tfdatasets`

Пакет `tfdatasets` можно использовать для создания эффективных конвейеров ввода для моделей машинного обучения. Его основным типом объекта является набор данных TensorFlow Dataset.

Объект Dataset является итерируемым: вы можете вызвать для него `as_iterator()`, чтобы создать итератор, а затем несколько раз вызвать `iter_next()` для итератора, чтобы сгенерировать последовательности данных. Чаще всего вы будете использовать объекты набора данных для создания пакетов входных данных и меток. Вы можете передать объект Dataset непосредственно методу `fit()` модели Keras.

Объект Dataset поддерживает многие ключевые функции, которые было бы сложно реализовать самостоятельно, в частности асинхронную предварительную выборку данных (предварительная обработка очередного пакета данных, в то время как предыдущий обрабатывается моделью, что обеспечивает непрерывное выполнение).

Пакет `tfdatasets` предоставляет функциональный API для изменения наборов данных. Вот простой наглядный пример: давайте создадим экземпляр Dataset из массива R целочисленной последовательности. Мы рассмотрим 100 выборок, где каждая выборка представляет собой век-

тор размера 6 (другими словами, наш исходный массив R представляет собой матрицу формы (100, 6)):

```
library(tfdatasets)
example_array <- array(seq(100*6), c(100, 6))
head(example_array)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1  101  201  301  401  501
[2,]    2  102  202  302  402  502
[3,]    3  103  203  303  403  503
[4,]    4  104  204  304  404  504
[5,]    5  105  205  305  405  505
[6,]    6  106  206  306  406  506
```

Функцию `tensor_slices_dataset()` можно использовать для создания набора данных TF из массива R или списка (необязательно именованного) массивов R

```
dataset <- tensor_slices_dataset(example_array) ←
```

Сначала наш набор данных дает только отдельные образцы:

```
dataset_iterator <- as_iterator(dataset)
for(i in 1:3) {
  element <- iter_next(dataset_iterator)
  print(element)
}
```

```
tf.Tensor([ 1 101 201 301 401 501], shape=(6), dtype=int32)
tf.Tensor([ 2 102 202 302 402 502], shape=(6), dtype=int32)
tf.Tensor([ 3 103 203 303 403 503], shape=(6), dtype=int32)
```

Обратите внимание, что итератор набора данных по умолчанию выдает тензоры TensorFlow. Обычно это то, что вам нужно, и это наиболее подходящий тип для таких методов, как `fit()`. Однако в некоторых ситуациях вы можете предпочесть, чтобы итератор вместо этого выдавал пакеты массивов R; в этой ситуации вы можете вызвать `as_array_iterator()` вместо `as_iterator()`:

```
dataset_array_iterator <- as_array_iterator(dataset)
for(i in 1:3) {
  element <- iter_next(dataset_array_iterator)
  str(element)
}
```

```
int [1:6(1d)] 1 101 201 301 401 501
int [1:6(1d)] 2 102 202 302 402 502
int [1:6(1d)] 3 103 203 303 403 503
```

Мы можем использовать `dataset_batch()` для пакетной обработки данных:

```
batched_dataset <- dataset %>%
  dataset_batch(3)
batched_dataset_iterator <- as_iterator(batched_dataset)
for(i in 1:3) {
```

```

element <- iter_next(batched_dataset_iterator)
print(element)
}

tf.Tensor(
[[ 1 101 201 301 401 501]
 [ 2 102 202 302 402 502]
 [ 3 103 203 303 403 503]], shape=(3, 6), dtype=int32)
tf.Tensor(
[[ 4 104 204 304 404 504]
 [ 5 105 205 305 405 505]
 [ 6 106 206 306 406 506]], shape=(3, 6), dtype=int32)
tf.Tensor(
[[ 7 107 207 307 407 507]
 [ 8 108 208 308 408 508]
 [ 9 109 209 309 409 509]], shape=(3, 6), dtype=int32)

```

В более широком смысле у нас есть доступ к целому ряду полезных методов набора данных, таких как:

- `dataset_shuffle(buffer_size)` – перемешивает элементы в буфере;
- `dataset_prefetch(buffer_size)` – выполняет предварительную выборку буфера элементов в памяти графического процессора для более эффективного использования устройства;
- `dataset_map(fn)` – применяет произвольное преобразование к каждому элементу набора данных (функция `fn`, которая принимает в качестве входных данных один элемент, полученный набором данных).

В частности, вы нередко будете использовать метод `dataset_map()`. В качестве примера мы будем использовать его для изменения элементов в нашем демонстрационном наборе данных из формы (6) в форму (2, 3):

```

reshaped_dataset <- dataset %>%
  dataset_map(function(element) tf$reshape(element, shape(2, 3)))

reshaped_dataset_iterator <- as_iterator(reshaped_dataset)
for(i in 1:3) {
  element <- iter_next(reshaped_dataset_iterator)
  print(element)
}

```

Обратите внимание, что `tf$reshape()` изменяет форму с использованием семантики стиля C (основной строки)

```

tf.Tensor(
[[ 1 101 201]
 [301 401 501]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 2 102 202]
 [302 402 502]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 3 103 203]
 [303 403 503]], shape=(2, 3), dtype=int32)

```

В этой главе вы увидите больше и другие примеры использования `dataset_map()`.

Рассмотрим вывод одного из таких объектов Dataset: он возвращает пакеты изображений 180×180 в формате RGB (с формой (32, 180, 180, 3)) и бинарные метки с формой (32).

#### Листинг 8.10 Отображение форм данных и меток, полученных из Dataset

```
c(data_batch, labels_batch) %<-% iter_next(as_iterator(train_dataset))
data_batch$shape
```

```
| TensorShape([32, 180, 180, 3])
```

```
labels_batch$shape
```

```
| TensorShape([32])
```

Обучим модель на нашем наборе данных. Мы будем использовать аргумент `validation_data` для метода `fit()`, чтобы отслеживать метрики проверки на разных объектах Dataset.

Важно отметить, что мы также будем использовать обратный вызов `callback_model_checkpoint()` для сохранения модели после каждой эпохи. Мы настроим его с помощью пути, указывающего, где сохранить файл, а также аргументов `save_best_only = TRUE` и `monitor = "val_loss"`: они говорят обратному вызову сохранять новый файл (перезаписывая любой имеющийся), только когда текущее значение метрики `val_loss` ниже, чем предыдущие значения. Это гарантирует, что ваш сохраненный файл всегда будет содержать состояние модели, соответствующее ее наиболее эффективной эпохе обучения с точки зрения точности на данных проверки. В результате нам не придется переобучать новую модель для меньшего количества эпох, если обнаружится переобучение: мы можем просто загрузить сохраненный файл.

#### Листинг 8.11 Обучение модели с использованием объекта Dataset

```
callbacks <- list(
  callback_model_checkpoint(
    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
  )
```

Построим графики изменения точности и потерь модели по обучающим и проверочным данным в процессе обучения (рис. 8.9).

### Листинг 8.12 Построение кривых потерь и точности в процессе обучения

```
plot(history)
```

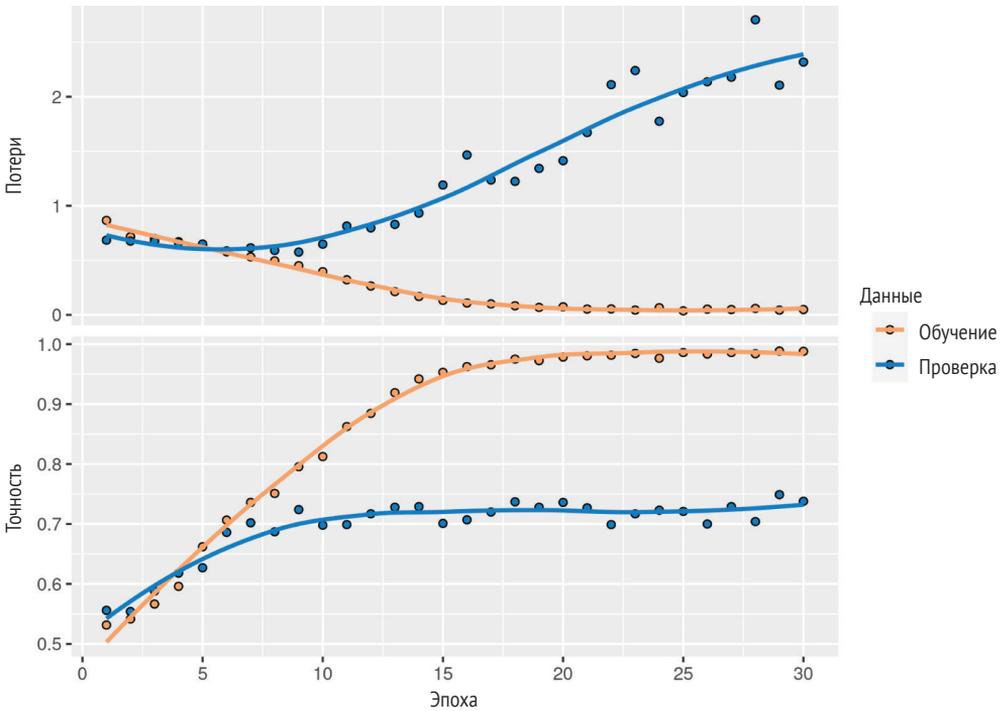


Рис. 8.9 Точность и потери на этапах обучения и проверки

На графиках четко наблюдается эффект переобучения. Точность на обучающих данных линейно растет и приближается к 100 %, тогда как точность на проверочных данных останавливается на отметке 75 %. Потери на этапе проверки достигают минимума всего после 10 эпох и затем начинают расти, а потери на этапе обучения продолжают линейно уменьшаться на протяжении всего периода обучения.

### Листинг 8.13 Оценка модели на контрольном наборе

```
test_model <- load_model_tf("convnet_from_scratch.keras")
result <- evaluate(test_model, test_dataset)
cat(sprintf("Test accuracy: %.3f\n", result["accuracy"]))
```

```
| Test accuracy: 0.740
```

Точность на контрольном наборе составляет 74 %. (Начальное состояние сети инициализируется случайными весами, поэтому ваш результат может незначительно отличаться.)

Поскольку у нас относительно немного обучающих образцов (2000), переобучение становится проблемой номер один. Вы уже знаете несколько методов, помогающих смягчить эту проблему, таких как прореживание и сокращение весов (L2-регуляризация). Теперь мы познакомимся с еще одним, характерным для распознавания образов и используемым почти повсеместно при обработке изображений с применением моделей глубокого обучения, – *расширением данных* (data augmentation).

## 8.2.5 Расширение данных

Причиной переобучения является недостаточное количество образцов для обучения модели, способной обобщать новые данные. Имея бесконечный объем данных, можно было бы получить модель, учитывающую все аспекты распределения данных: эффект переобучения никогда не наступил бы. Прием расширения реализует подход создания дополнительных обучающих данных из имеющихся путем трансформации образцов множеством случайных преобразований, дающих правдоподобные изображения. Цель расширения данных 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.

В Keras это можно сделать, добавив несколько *слоев расширения данных* в начале вашей модели. Давайте начнем с примера: слой `keras_model_sequential()` связывает несколько случайных преобразований изображения. В нашей модели мы вставили его прямо перед `layer_rescaling()`.

### Листинг 8.14 Определение этапа расширения данных для добавления в модель расширения изображений

```
data_augmentation <- keras_model_sequential() %>%  
  layer_random_flip("horizontal") %>%  
  layer_random_rotation(0.1) %>%  
  layer_random_zoom(0.2)
```

Здесь представлена лишь часть возможных вариантов (полный список вы найдете в документации к фреймворку Keras). Давайте бегло рассмотрим этот код:

- `layer_random_flip("horizontal")` – применяет горизонтальное отражение к случайным 50 % проходящих через него изображений;
- `layer_random_rotation(0.1)` – поворачивает входные изображения на случайную величину в диапазоне  $[-10\%, +10\%]$  (это доли полного круга, в градусах диапазон составляет  $[-36, +36]$ );
- `layer_random_zoom(0.2)` – увеличивает или уменьшает изображение на случайный коэффициент в диапазоне  $[-20\%, +20\%]$ .

На рис. 8.10 представлены случайные изображения, полученные путем рандомизированного расширения данных (листинг 8.15).

### Листинг 8.15 Отображение некоторых обучающих изображений, подвергшихся случайным преобразованиям

```

library(tfdatasets)
batch <- train_dataset %>%
  as_iterator() %>%
  iter_next()

c(images, labels) %<-% batch

par(mfrow = c(3, 3), mar = rep(.5, 4))
image <- images[1, , , ]
plot(as.raster(as.array(image), max = 255))

for (i in 2:9) {
  augmented_images <- data_augmentation(images)
  augmented_image <- augmented_images[1, , , ]
  plot(as.raster(as.array(augmented_image), max = 255))
}

```

Подготовка графического устройства для девяти изображений

Построение первого изображения из партии (без расширения)

Отображение первого изображения в выходном пакете. Для каждой из восьми итераций это разное расширение одного и того же изображения

Применение шага расширения к пакету изображений

Если мы обучим новую модель, используя эту конфигурацию увеличения данных, модель никогда не увидит одни и те же входные данные дважды. Но входные данные, которые она видит, по-прежнему сильно взаимосвязаны, потому что они исходят из небольшого количества исходных изображений, – мы не можем производить новую информацию, а лишь модифицируем имеющиеся данные. Этого может быть недостаточно, чтобы полностью избавиться от переобучения. Чтобы еще успешнее противостоять переобучению, мы также добавим в нашу модель слой `layer_dropout()` прямо перед плотно связанным классификатором.

И последнее, что вы должны знать о слоях рандомизированного расширения изображений: как и `layer_dropout()`, они неактивны во время логического вывода, когда мы вызываем `predict()` или `evaluate()`. В рабочем режиме наша модель будет вести себя точно так же, как если бы она не содержала слоев расширения и прореживания данных.

### Листинг 8.16 Определение новой сверточной нейронной сети, включающей уровни расширения и прореживания

```

inputs <- layer_input(shape = c(180, 180, 3))
outputs <- inputs %>%
  data_augmentation() %>%
  layer_rescaling(1 / 255) %>%
  layer_conv_2d(filters = 32, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 64, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_conv_2d(filters = 128, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%

```

```
layer_conv_2d(filters = 256, kernel_size = 3, activation = "relu") %>%  
layer_max_pooling_2d(pool_size = 2) %>%  
layer_conv_2d(filters = 256, kernel_size = 3, activation = "relu") %>%  
layer_flatten() %>%  
layer_dropout(0.5) %>%  
layer_dense(1, activation = "sigmoid")  
  
model <- keras_model(inputs, outputs)  
  
model %>% compile(loss = "binary_crossentropy",  
                 optimizer = "rmsprop",  
                 metrics = "accuracy")
```



Рис. 8.10 Изображения собаки, полученные путем рандомизированного расширения данных

Обучим сеть, задействовав расширение данных и прореживание. Поскольку мы ожидаем, что переобучение произойдет намного позже, зададим продолжительность обучения в три раза больше – сто эпох (листинг 8.17).

## Листинг 8.17 Обучение регуляризованной сверточной нейронной сети

```

callbacks <- list(
  callback_model_checkpoint(
    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
)

```

Снова построим график функции потерь модели (рис. 8.11). Благодаря дополнению и отбрасыванию данных переобучение начинается намного позже, примерно после 60–70 эпох (по сравнению с 10-й эпохой для исходной модели). Точность модели на проверочных данных постоянно находится в диапазоне 80–85 % – заметное улучшение по сравнению с нашей первой попыткой:

```
plot(history)
```

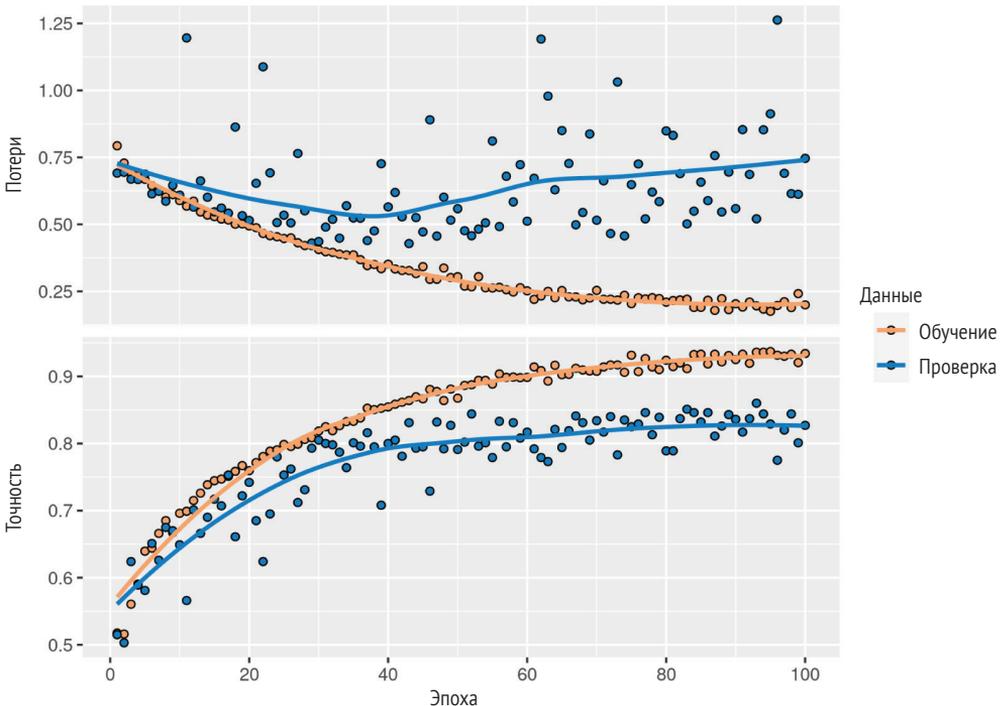


Рис. 8.11 Потери модели при обучении и проверке с использованием расширения данных

Проверим точность модели на контрольных данных (листинг 8.18).

#### Листинг 8.18 Проверка модели на контрольных данных

```
test_model <- load_model_tf("convnet_from_scratch_with_augmentation.keras")
result <- evaluate(test_model, test_dataset)
cat(sprintf("Test accuracy: %.3f\n", result["accuracy"]))
```

```
| Test accuracy: 0.814
```

Мы получаем точность теста 81,4 %. Наша модель работает все лучше! Если вы запускаете примеры кода на своем компьютере, не удаляйте сохраненный файл (`convnet_from_scratch_with_augmentation.keras`), потому что мы будем использовать его для экспериментов в следующей главе.

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 этой задачи заключается в использовании предварительно обученной модели, но об этом мы поговорим в следующих двух разделах.

## 8.3 Использование предварительно обученной сверточной нейронной сети

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 этот исходный набор данных довольно велик и достаточно обобщен, тогда пространственная иерархия признаков, выделенных сетью, может эффективно выступать в роли обобщенной модели видимого мира и быть полезной во многих разных задачах распознавания образов, даже если эти новые задачи будут связаны с совершенно иными классами, отличными от классов в оригинальной задаче. Другими словами, можно обучить сеть на изображениях из ImageNet (где подавляющее большинство классов – животные и бытовые предметы) и затем использовать эту обученную сеть для идентификации чего-то иного, например предметов мебели на изображениях. Такая переносимость изученных признаков между разными задачами – главное преимущество глубокого обучения перед многими более старыми приемами поверхностного обучения, что делает

глубокое обучение очень эффективным инструментом для решения задач с малым объемом данных.

В нашем случае мы возьмем за основу сверточную нейронную сеть, обученную на наборе ImageNet (1,4 млн изображений, разбитых на 1000 разных классов). Коллекция ImageNet содержит множество изображений разных животных, включая разновидности кошек и собак, а значит, можно рассчитывать, что модель, обученная на этой коллекции, прекрасно справится с нашей задачей классификации изображений кошек и собак.

Мы воспользуемся архитектурой VGG16, разработанной Кареном Симоняном (Karen Simonyan) и Эндрю Циссерманом (Andrew Zisserman) в 2014 году<sup>1</sup>. Хотя это довольно старая модель, далеко отставшая от современного уровня, и к тому же немного тяжелее многих более современных моделей, я выбрал ее, потому что ее архитектура похожа на примеры, представленные в этой книге раньше, и вам будет легче понять последующие примеры кода без дополнительных пояснений. Возможно, это ваша первая встреча с одним из представителей всех этих моделей, названия которых вызывают дрожь, – VGG, ResNet, Inception, Inception-ResNet, Xception и т. д.; но со временем вы привыкнете к ним, потому что они часто будут встречаться на вашем пути, если вы продолжите заниматься применением глубокого обучения в распознавании образов.

Есть два приема использования предварительно обученных сетей: *выделение признаков* (feature extraction) и *дообучение* (fine-tuning). Мы рассмотрим оба и начнем с выделения признаков.

### 8.3.1 Выделение признаков

Выделение признаков заключается в использовании представлений, изученных предыдущей сетью, для выделения признаков из новых образцов, которые затем пропускаются через новый классификатор, обучаемый с нуля.

Как было показано выше, сверточные нейронные сети, используемые для классификации изображений, состоят из двух частей: они начинаются с последовательности слоев выбора значений и свертки и заканчиваются полносвязным классификатором. Первая часть называется *сверточной основой* (convolutional base) модели. В случае со сверточными нейронными сетями процесс выделения признаков заключается в том, чтобы взять сверточную основу предварительно обученной сети, пропустить через нее новые данные и на основе вывода обучить новый классификатор (рис. 8.12).

Почему повторно используется только сверточная основа? Нельзя ли повторно использовать полносвязный классификатор? В общем случае этого следует избегать. Причина в том, что представления,

---

<sup>1</sup> Karen Simonyan, Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv (2014), <https://arxiv.org/abs/1409.1556>.

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.

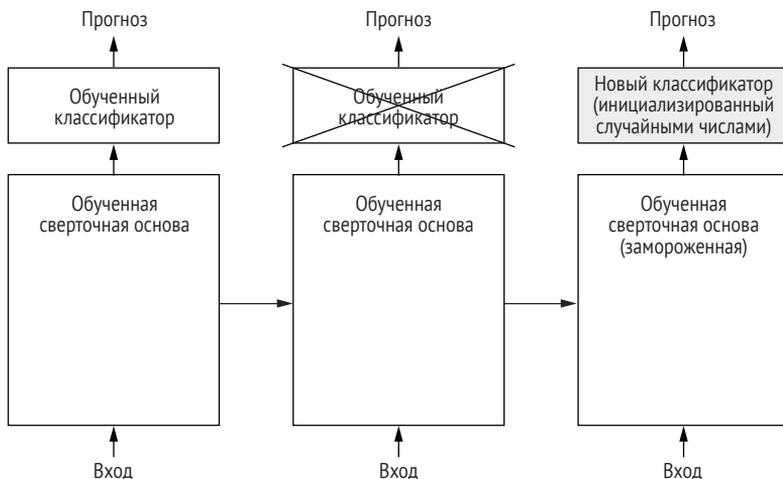


Рис. 8.12 Замена классификаторов при использовании одной и той же сверточной основы

Отметим также, что уровень обобщенности (и, соответственно, пригодности к повторному использованию) представлений, выделенных конкретными сверточными слоями, зависит от глубины слоя в модели. Слои, расположенные первыми, выделяют локальные, наиболее обобщенные карты признаков (таких как визуальные границы, цвет и текстура), тогда как слои, следующие за ними, выделяют более абстрактные понятия (такие как «глаз кошки» или «глаз собаки»). Поэтому если новый набор данных существенно отличается от набора, на котором обучалась оригинальная модель, возможно, большего успеха можно добиться, если использовать только несколько первых слоев модели, а не всю сверточную основу.

В нашем случае, поскольку набор классов ImageNet содержит несколько классов кошек и собак, вероятно, было бы полезно повторно

использовать информацию, содержащуюся в полносвязных слоях оригинальной модели. Но мы не будем этого делать, чтобы охватить более общий случай, когда набор классов из новой задачи не пересекается с набором классов оригинальной модели. А теперь перейдем к практике и используем сверточную основу сети VGG16, обученной на данных ImageNet, для выделения полезных признаков из изображений кошек и собак, а затем обучим классификатор кошек и собак, опираясь на эти признаки.

Модель VGG16 в числе прочих входит в состав фреймворка Keras. Все они экспортированы как функции с префиксом `application_`. Вот список моделей классификации изображений (все они предварительно обучены на наборе ImageNet), доступных в Keras:

- Xception;
- ResNet;
- MobileNet;
- EfficientNet;
- DenseNet;
- ...и др.

Создадим экземпляр модели VGG16 (листинг 8.19).

#### Листинг 8.19 Создание экземпляра сверточной основы VGG16

```
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(180, 180, 3)
)
```

Здесь функции передаются три аргумента:

- `weights` определяет источник весов для инициализации модели;
- `include_top` определяет необходимость подключения к сети полносвязного классификатора. По умолчанию этот классификатор соответствует 1000 классов из ImageNet. Так как мы намереваемся использовать свой полносвязный классификатор (только с двумя классами: `cat` и `dog`), мы не будем подключать его;
- `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. Здесь мы передаем его, чтобы иметь возможность визуализировать в сводке модели, как размер карт объектов уменьшается с каждым новым слоем свертки и объединения.

Далее приводится информация об архитектуре сверточной основы VGG16. Она напоминает уже знакомые вам простые сверточные нейронные сети:

conv\_base

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_5 (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 поместим полносвязный классификатор.

Далее можно пойти двумя путями:

- пропустить наш набор данных через сверточную основу, записать получившийся массив на диск и затем использовать его как входные данные для отдельного полносвязного классификатора, похожего на тот, что мы видели в первой части книги. Это быстрое и недорогое решение, потому что оно требует запускать сверточную основу только один раз для каждого входного изображения, а сверточная основа – самая дорогостоящая часть конвейера. Но по той же причине этот подход не позволит использовать прием расширения данных;
- дополнить имеющуюся модель (conv\_base) полносвязными слоями Dense и пропустить все входные данные. Этот подход позволяет использовать расширение данных, потому что каждое изображение проходит через сверточную основу каждый раз, когда попадает в модель. Но по той же причине этот подход намного дороже первого в вычислительном отношении.

Мы изучим оба подхода. Сначала рассмотрим код, реализующий первый вариант: запись вывода `conv_base` в ответ на входные данные и использование этого вывода в качестве входных данных новой модели.

### БЫСТРОЕ ВЫДЕЛЕНИЕ ПРИЗНАКОВ БЕЗ РАСШИРЕНИЯ ДАННЫХ

Мы начнем с извлечения признаков в виде массивов R, вызвав метод `predict()` модели `conv_base` для наших наборов данных для обучения, проверки и контроля.

Итеративно пройдем по нашим наборам данных, чтобы выделить признаки VGG16.

#### Листинг 8.20 Выделение признаков VGG16 и соответствующих меток

```
get_features_and_labels <- function(dataset) {
  n_batches <- length(dataset)
  all_features <- vector("list", n_batches)
  all_labels <- vector("list", n_batches)
  iterator <- as_array_iterator(dataset)
  for (i in 1:n_batches) {
    c(images, labels) %<-% iter_next(iterator)
    preprocessed_images <- imagenet_preprocess_input(images)
    features <- conv_base %>% predict(preprocessed_images)

    all_labels[[i]] <- labels
    all_features[[i]] <- features
  }

  all_features <- listarrays::bind_on_rows(all_features)
  all_labels <- listarrays::bind_on_rows(all_labels)
}

list(all_features, all_labels)

c(train_features, train_labels) %<-% get_features_and_labels(train_dataset)
c(val_features, val_labels) %<-% get_features_and_labels(validation_dataset)
c(test_features, test_labels) %<-% get_features_and_labels(test_dataset)
```

Объединение списка массивов R по первой оси (измерение пакетов)

Важно отметить, что `predict()` ожидает только изображения, а не метки, но наш текущий набор данных дает пакеты, которые содержат как изображения, так и их метки. Более того, модель VGG16 предполагает, что входные данные предварительно обработаны с помощью функции `imagenet_preprocess_input()`, которая масштабирует зна-

чения пикселей до соответствующего диапазона. Извлеченные признаки в настоящее время имеют форму (samples, 5, 5, 512):

```
dim(train_features)
```

```
[1] 2000 5 5 512
```

На этом этапе можно определить наш полносвязный классификатор (обратите внимание на использование прореживания для регуляризации) и обучить его на только что записанных данных и метках.

### Листинг 8.21 Определение и обучение полносвязного классификатора

```
inputs <- layer_input(shape = c(5, 5, 512))
outputs <- inputs %>%
  layer_flatten() %>% ← | Обратите внимание на использование
  layer_dense(256) %>% | layer_flatten() перед передачей
  layer_dropout(.5) %>% | признаков в layer_dense()
  layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>% compile(loss = "binary_crossentropy",
  optimizer = "rmsprop",
  metrics = "accuracy")

callbacks <- list(
  callback_model_checkpoint(
    filepath = "feature_extraction.keras",
    save_best_only = TRUE,
    monitor = "val_loss"
  )
)

history <- model %>% fit(
  train_features, train_labels,
  epochs = 20,
  validation_data = list(val_features, val_labels),
  callbacks = callbacks
)
```

Обучение проходит очень быстро, потому что мы определили только два полносвязных уровня – одна эпоха длится меньше одной секунды, даже при выполнении на CPU.

Посмотрим теперь на графики изменения потерь и точности в процессе обучения (листинг 8.22 и рис. 8.13).

## Листинг 8.22 Построение графиков

```
plot(history)
```

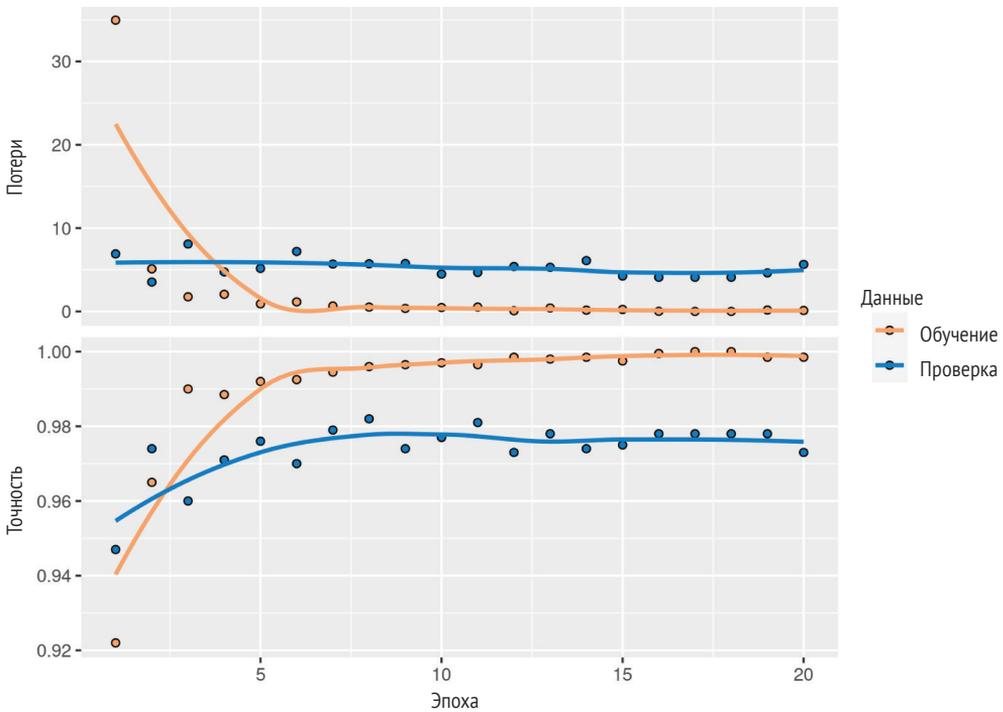


Рис. 8.13 Точность и потери на этапах проверки и обучения для простого извлечения признаков

Мы достигли точности, близкой к 97 %, – значительно более высокой, чем в предыдущем разделе, где обучали небольшую модель с нуля. Это немного некорректное сравнение, потому что ImageNet содержит много экземпляров собак и кошек, а это означает, что наша предварительно обученная модель уже имеет точные знания, необходимые для выполнения поставленной задачи. Так будет не во всех случаях, когда вы используете предварительно обученные признаки.

Но графики также показывают, что почти с самого начала стал проявляться эффект переобучения, несмотря на выбор довольно большого коэффициента прореживания. Это объясняется тем, что данный прием не использует расширение данных, которое необходимо для предотвращения переобучения на небольших наборах изображений.

### ВЫДЕЛЕНИЕ ПРИЗНАКОВ С РАСШИРЕНИЕМ ДАННЫХ

Теперь рассмотрим второй прием выделения признаков, более медленный и дорогостоящий, но позволяющий использовать расшире-

ние данных в процессе обучения: создание модели, которая связывает conv\_base с новым полносвязным классификатором, и обучение ее от начала до конца на входных данных.

Перед компиляцией и обучением модели очень важно *заморозить сверточную основу*. Замораживание одного или нескольких слоев предотвращает изменение весовых коэффициентов в них в процессе обучения. Если этого не сделать, тогда представления, прежде изученные сверточной основой, изменятся в процессе обучения на новых данных. Так как полносвязные слои сверху инициализируются случайными значениями, в сети могут произойти существенные изменения весов, фактически разрушив представления, полученные ранее. В Keras, чтобы заморозить сеть, нужно вызвать функцию freeze\_weights().

### Листинг 8.23 Создание и замораживание сверточной основы VGG16

```
conv_base <- application_vgg16(  
  weights = "imagenet",  
  include_top = FALSE)  
freeze_weights(conv_base)
```

Вызов freeze\_weights() очищает список весов, подлежащих обучению в слое или модели.

### Листинг 8.24 Распечатка списка обучаемых весов до и после замораживания

```
unfreeze_weights(conv_base)  
cat("This is the number of trainable weights",  
  "before freezing the conv base:",  
  length(conv_base$trainable_weights), "\n")
```

```
| This is the number of trainable weights before freezing the conv base: 26
```

```
freeze_weights(conv_base)  
cat("This is the number of trainable weights",  
  "after freezing the conv base:",  
  length(conv_base$trainable_weights), "\n")
```

```
| This is the number of trainable weights after freezing the conv base: 0
```

Теперь мы можем создать новую модель, которая объединяет:

- 1 этап дополнения данных;
- 2 нашу замороженную сверточную основу;
- 3 полносвязный классификатор.

```
data_augmentation <- keras_model_sequential() %>%  
  layer_random_flip("horizontal") %>%  
  layer_random_rotation(0.1) %>%  
  layer_random_zoom(0.2)
```

```

inputs <- layer_input(shape = c(180, 180, 3))
outputs <- inputs %>%
  data_augmentation() %>% ← Применим дополнение данных
  imagenet_preprocess_input() %>% ← Применим масштабирование
  conv_base() %>%           входного значения
  layer_flatten() %>%
  layer_dense(256) %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>% compile(loss = "binary_crossentropy",
                 optimizer = "rmsprop",
                 metrics = "accuracy")

```

При такой конфигурации будут обучаться только веса из двух добавленных полносвязных слоев. Всего получается четыре весовых тензора: по два на слой (основная матрица весов и вектор смещения). Учтите, чтобы эти изменения вступили в силу, вы должны сначала скомпилировать модель. Если вы когда-либо изменяли обучение весов после компиляции, вам следует перекомпилировать модель, иначе эти изменения будут проигнорированы.

Давайте обучим нашу модель. Благодаря расширению данных модели потребуется гораздо больше времени, чтобы достичь переобучения, поэтому мы можем выполнить обучение на большем количестве эпох – пусть их будет 50.

Этот прием настолько дорогостоящий, что его следует применять только при наличии GPU – он абсолютно не под силу CPU. Если у вас нет возможности запустить свой код на GPU, первый путь остается для вас единственным доступным решением:

```

callbacks <- list(
  callback_model_checkpoint(
    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 потерь и точности в процессе обучения (рис. 8.14). Как видите, мы достигли точности более 98 % на этапе проверки. Этот результат намного лучше по сравнению с предыдущей моделью.

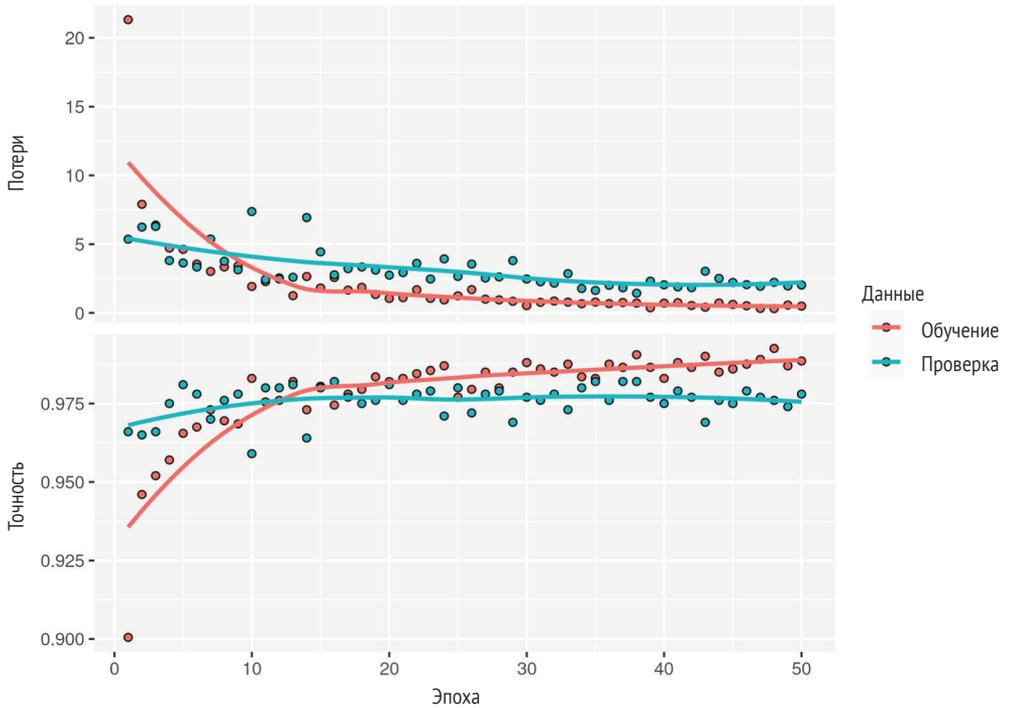


Рис. 8.14 Точность и потери на этапах проверки и обучения для извлечения признаков с расширением данных

Проверим модель на контрольных данных (листинг 8.25).

#### Листинг 8.25 Оценка модели на контрольном наборе

```
test_model <- load_model_tf(  
    "feature_extraction_with_data_augmentation.keras")  
result <- evaluate(test_model, test_dataset)  
cat(sprintf("Test accuracy: %.3f\n", result["accuracy"]))
```

Test accuracy: 0.977

Мы получили точность на контрольных данных 97,7 %. Это лишь незначительное улучшение по сравнению с точностью предыдущей версии модели, что немного разочаровывает, учитывая хорошие результаты на проверочных данных. Точность модели всегда зависит от набора образцов, на которых вы ее оцениваете. Некоторые наборы образцов могут быть более сложными, чем другие, и хорошие результаты на одном наборе не обязательно будут полностью повторяться на других наборах.

### 8.3.2 Дообучение ранее обученной модели

Другой широко используемый прием повторного использования модели, дополняющий выделение признаков, – дообучение (fine-tuning) (рис. 8.15). Дообучение заключается в размораживании нескольких

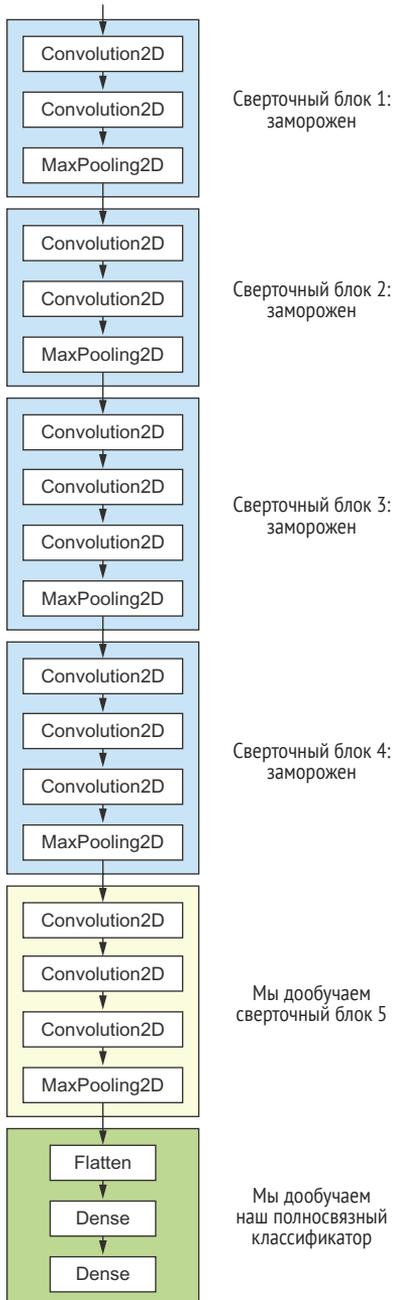


Рис. 8.15 Дообучение последнего сверточного блока сети VGG16

верхних уровней замороженной модели, использовавшейся для выделения признаков, и совместном обучении вновь добавленной части модели (в данном случае полносвязного классификатора) и этих верхних уровней. Этот прием называется дообучением, потому что немного корректирует наиболее абстрактные представления в повторно используемой модели, чтобы сделать их более актуальными для данной задачи.

Выше я отмечал, что необходимо заморозить сверточную основу сети VGG16, чтобы получить возможность обучить классификатор, инициализированный случайными значениями. По той же причине после обучения классификатора можно дообучить несколько верхних слоев сверточной основы. Если классификатор еще не обучен, сигнал ошибки, распространяющийся по сети в процессе дообучения, окажется слишком велик, и представления, полученные на предыдущем этапе обучения, будут разрушены. Поэтому для дообучения модели необходимо 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 вам не следует размораживать слои пакетной нормализации, которые здесь неуместны, поскольку в VGG16 таких слоев нет. Пакетная нормализация и ее влияние на дообучение объясняются в следующей главе);
- 5 дообучить эти слои и добавленную часть вместе.

Мы уже выполнили первые три шага в ходе выделения признаков. Теперь выполним шаг 4: разморозим `conv_base` и заморозим отдельные слои в ней.

Вспомним, что наша сверточная основа выглядит примерно следующим образом:

```
conv_base
```

```
Model: "vgg16"
```

Layer (type)	Output Shape	Param #	Trainable
input_7 (InputLayer)	[(None, None, None, 3)]	0	N
block1_conv1 (Conv2D)	(None, None, None, 64)	1792	N
block1_conv2 (Conv2D)	(None, None, None, 64)	36928	N
block1_pool (MaxPooling2D)	(None, None, None, 64)	0	N
block2_conv1 (Conv2D)	(None, None, None, 128)	73856	N
block2_conv2 (Conv2D)	(None, None, None, 128)	147584	N
block2_pool (MaxPooling2D)	(None, None, None, 128)	0	N
block3_conv1 (Conv2D)	(None, None, None, 256)	295168	N
block3_conv2 (Conv2D)	(None, None, None, 256)	590080	N
block3_conv3 (Conv2D)	(None, None, None, 256)	590080	N
block3_pool (MaxPooling2D)	(None, None, None, 256)	0	N
block4_conv1 (Conv2D)	(None, None, None, 512)	1180160	N

```

block4_conv2 (Conv2D)      (None, None, None, 512) 2359808 N
block4_conv3 (Conv2D)      (None, None, None, 512) 2359808 N
block4_pool (MaxPooling2D) (None, None, None, 512) 0 N
block5_conv1 (Conv2D)      (None, None, None, 512) 2359808 N
block5_conv2 (Conv2D)      (None, None, None, 512) 2359808 N
block5_conv3 (Conv2D)      (None, None, None, 512) 2359808 N
block5_pool (MaxPooling2D) (None, None, None, 512) 0 N
=====
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688

```

Мы дообучим последние три сверточных слоя, что означает, что все слои до `block4_pool` должны быть заморожены, а слои `block5_conv1`, `block5_conv2` и `block5_conv3` должны остаться обучаемыми.

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 уровни кодируют более конкретные признаки. Намного полезнее дополнительно выделить 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 млн параметров, поэтому было бы слишком рискованно пытаться дообучить ее целиком на нашем небольшом наборе данных.

Следовательно, в данной ситуации лучшей стратегией будет дообучить только несколько верхних слоев сверточной основы. Сделаем это, начав с того места, на котором мы остановились в предыдущем примере (листинг 8.26).

### Листинг 8.26 Замораживание всех слоев до четвертого от последнего

```

unfreeze_weights(conv_base, from = -4) ← from -4 – это сокращение от
conv_base                               length(conv_base$layers) + 1 - 4

```

```
Model: "vgg16"
```

Layer (type)	Output Shape	Param #	Trainable
input_1 (InputLayer)	[(None, None, None, 3)]	0	N
block1_conv1 (Conv2D)	(None, None, None, 64)	1792	N
block1_conv2 (Conv2D)	(None, None, None, 64)	36928	N
block1_pool (MaxPooling2D)	(None, None, None, 64)	0	N

block2_conv1 (Conv2D)	(None, None, None, 128)	73856	N
block2_conv2 (Conv2D)	(None, None, None, 128)	147584	N
block2_pool (MaxPooling2D)	(None, None, None, 128)	0	N
block3_conv1 (Conv2D)	(None, None, None, 256)	295168	N
block3_conv2 (Conv2D)	(None, None, None, 256)	590080	N
block3_conv3 (Conv2D)	(None, None, None, 256)	590080	N
block3_pool (MaxPooling2D)	(None, None, None, 256)	0	N
block4_conv1 (Conv2D)	(None, None, None, 512)	1180160	N
block4_conv2 (Conv2D)	(None, None, None, 512)	2359808	N
block4_conv3 (Conv2D)	(None, None, None, 512)	2359808	N
block4_pool (MaxPooling2D)	(None, None, None, 512)	0	N
block5_conv1 (Conv2D)	(None, None, None, 512)	2359808	Y
block5_conv2 (Conv2D)	(None, None, None, 512)	2359808	Y
block5_conv3 (Conv2D)	(None, None, None, 512)	2359808	Y
block5_pool (MaxPooling2D)	(None, None, None, 512)	0	Y

```

=====
Total params: 14,714,688
Trainable params: 7,079,424
Non-trainable params: 7,635,264

```

Теперь можно начинать дообучение сети. Для этого используем оптимизатор RMSProp с очень маленькой скоростью обучения. Причина использования низкой скорости обучения заключается в необходимости ограничить величину изменений, вносимых в представления трех дообучаемых уровней. Слишком большие изменения могут повредить эти представления.

### Листинг 8.27 Дообучение модели

```

model %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = "accuracy"
)

callbacks <- list(
  callback_model_checkpoint(
    filepath = "fine_tuning.keras",
    save_best_only = TRUE,
    monitor = "val_loss"
  )
)

history <- model %>% fit(
  train_dataset,
  epochs = 30,
  validation_data = validation_dataset,
  callbacks = callbacks
)

```

Теперь, наконец, можно оценить модель на контрольных данных:

```
model <- load_model_tf("fine_tuning.keras")
result <- evaluate(model, test_dataset)
cat(sprintf("Test accuracy: %.3f\n", result["accuracy"]))
```

```
| Test accuracy: 0.985
```

Здесь мы получили точность на уровне 98,5 % (ваш результат может отличаться примерно на один процент). В оригинальном состязании на сайте Kaggle, основанном на этом наборе данных, это был бы один из лучших результатов. Однако это не совсем справедливое сравнение, потому что мы использовали предварительно обученные признаки, которые уже содержат готовые знания о кошках и собаках, недоступные участникам соревнований в то время.

С другой стороны, благодаря современным методам глубокого обучения нам удалось достичь такого результата, используя лишь малую часть имеющихся обучающих данных (около 10 %). Между обучением на 20 000 и на 2000 образцах огромная разница!

Теперь у вас имеется 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. Вполне можно обучить такую сеть с нуля на очень небольшом наборе данных и получить приличный результат.
- Сверточные сети работают, изучая иерархию модульных шаблонов и концепций для представления визуального мира.
- Когда объем данных ограничен, главной проблемой становится переобучение. Расширение данных – мощное средство борьбы с переобучением при работе с изображениями.
- Существующую сверточную нейронную сеть 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.

# Глубокое обучение для компьютерного зрения

---

## **Эта глава охватывает следующие темы:**

- различные области компьютерного зрения: классификация изображений, сегментация изображений и обнаружение объектов;
- современные архитектуры сверточных сетей: остаточные связи, пакетная нормализация и свертки с разделением по глубине;
- методы визуализации и интерпретации знаний сверточной сети.

В предыдущей главе вы познакомились с базовыми моделями компьютерного зрения в виде последовательности слоев `layer_conv_2d()` и `layer_max_pooling_2d()` и простым вариантом их использования (бинарная классификация изображений). Но компьютерное зрение – это намного больше, чем классификация изображений! В этой главе мы подробно рассмотрим более разнообразные применения и передовые методы компьютерного зрения.

## 9.1 Три основные задачи компьютерного зрения

До сих пор мы рассматривали простейшие модели классификации изображений: на вход сети подаются изображение, на выходе появляется метка: «Это изображение, вероятно, содержит кошку; а на этом, вероятно, есть собака». Но классификация изображений – лишь одно из нескольких возможных применений глубокого обучения в компьютерном зрении. В целом есть три основные задачи компьютерного зрения, о которых вам нужно знать:

- *классификация изображения* – целью является присвоение изображению одной или нескольких меток. Это может быть классификация по одной метке (изображение может принадлежать только одной категории, исключая другие) или по нескольким меткам (маркировка всех категорий, к которым принадлежит изображение, как показано на рис. 9.1). Например, выполняя поиск по ключевому слову в приложении Google Photos, фактически вы обращаетесь с запросом к очень большой модели классификации с несколькими метками – более чем 20 000 различных классов, обученных на миллионах изображений;
- *сегментация изображения* – цель состоит в том, чтобы «сегментировать», или «разделить», изображение на разные области, причем каждая область обычно представляет категорию (как показано на рис. 9.1). Например, когда приложения для видеоконференций отображают пользовательский фон позади вас в ходе видеозвонка, они используют модель сегментации изображения, чтобы с точностью до пикселя отличить ваше лицо от фона;
- *обнаружение объектов* – цель состоит в том, чтобы нарисовать прямоугольники (называемые *ограничивающими рамками*) вокруг интересующих объектов на изображении и связать каждый прямоугольник с классом. Беспилотный автомобиль может использовать модель обнаружения объектов, например для наблюдения за автомобилями, пешеходами и знаками в поле зрения его камер.

Глубокое обучение для компьютерного зрения также включает в себя ряд несколько более узких задач, помимо трех вышеупомянутых. Сюда относятся оценка схожести изображений (оценка того, насколько визуально похожи два изображения), обнаружение ключевых точек (определение интересующих атрибутов изображения, таких как черты лица), определение позы, отрисовка 3D-сетки и т. д. Но классификация, сегментация и обнаружение составляют основу, с которой должен быть знаком каждый специалист по машинному обучению. Большинство приложений компьютерного зрения сводятся к одному из этих трех направлений.

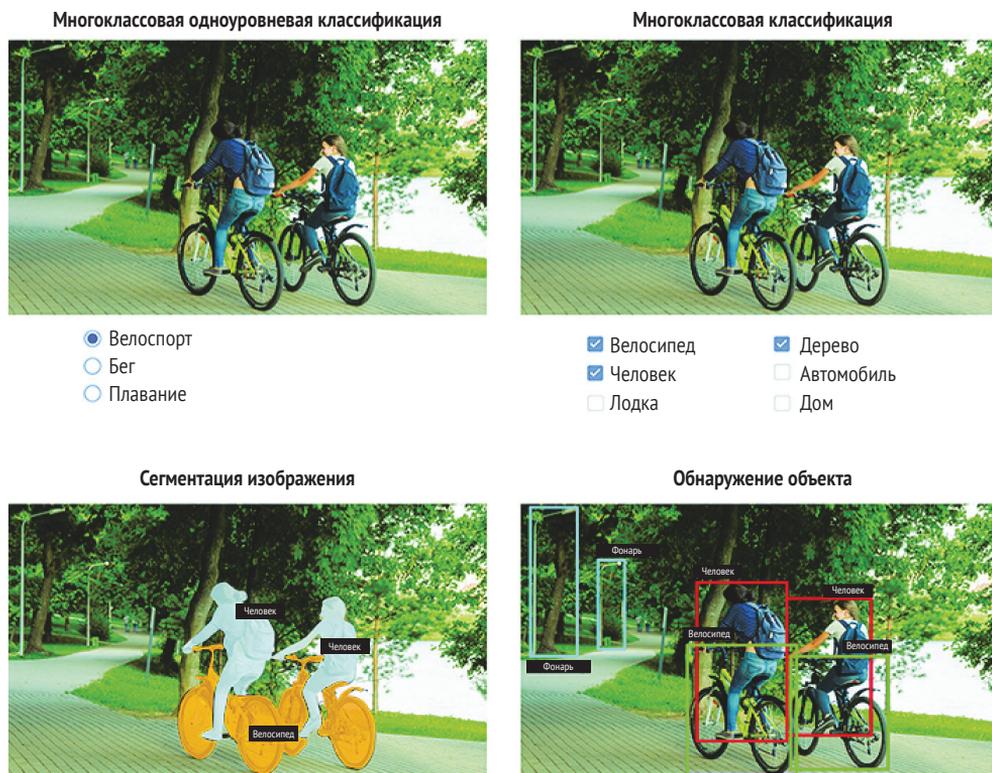


Рис. 9.1 Три основные задачи компьютерного зрения: классификация, сегментация, обнаружение

Вы видели классификацию изображений в действии в предыдущей главе. Далее мы займемся изучением сегментации. Это очень полезная и универсальная технология, и вдобавок вы можете применить знания, которые уже усвоили из предыдущих глав.

Следует сразу отметить, что мы не будем рассматривать обнаружение объектов, потому что это слишком специализированная и сложная тема для вводной книги. Однако вы можете ознакомиться с примером RetinaNet на сайте <https://tensorflow.rstudio.com/examples/>, в котором показано, как создать и обучить модель обнаружения объектов с нуля в R с помощью Keras.

## 9.2 Пример сегментации изображения

Сегментация изображения с помощью глубокого обучения заключается в использовании модели для присвоения класса каждому пикселю изображения. Путем классификации пикселей происходит разделение изображения на различные зоны (такие как «фон» и «передний план» или «дорога», «автомобиль» и «тротуар»). Существует

целая группа методов сегментации, которые можно использовать в области редактирования изображений и видео, автономного вождения, робототехники, медицинской визуализации и т. д. Есть два основных варианта сегментации изображений, о которых вам следует знать:

- *семантическая сегментация*, при которой каждый пиксель независимо классифицируется по семантической категории, например «кошка». Если на изображении две кошки, все соответствующие пиксели сопоставляются с одной и той же общей категорией «кошка» (рис. 9.2);
- *сегментация экземпляров*, которая направлена не только на классификацию пикселей изображения по категориям, но и на выделение отдельных экземпляров объектов. В изображении с двумя кошками сегментация экземпляров будет рассматривать «кошку 1» и «кошку 2» как два отдельных класса пикселей (рис. 9.2).

В этом примере мы сосредоточимся на семантической сегментации: мы еще раз возьмем изображения кошек и собак и на этот раз научим нейросеть различать основной объект и его фон.

Мы будем работать с набором данных Oxford-IIIТ Pets (<https://www.robots.ox.ac.uk/~vgg/data/pets/>), который содержит 7390 изображений различных пород кошек и собак вместе с масками сегментации переднего и заднего планов для каждого изображения. *Маска сегментации* – это эквивалент метки для сегментации изображения, т. е. изображение того же размера, что и входное изображение, с одним цветовым каналом, где каждое целочисленное значение соответствует классу соответствующего пикселя во входном изображении. В нашем случае пиксели наших масок сегментации могут принимать одно из трех целочисленных значений:

- 1 (передний план);
- 2 (фон);
- 3 (контур).

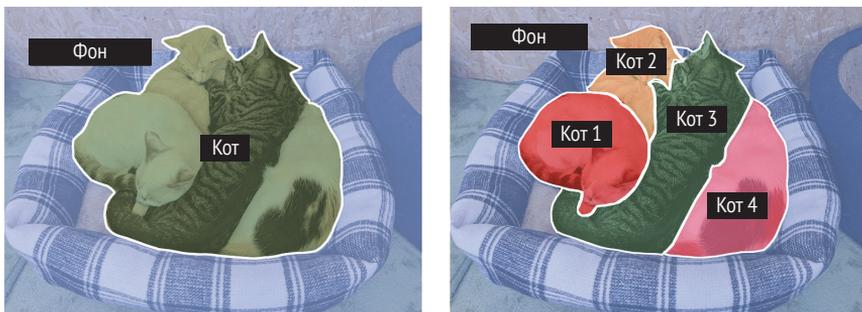


Рис. 9.2 Различие между типами сегментации

Начнем с загрузки и распаковки нашего набора данных, используя утилиты `download.file()` и `untar()`, входящие в состав R. Как и в главе 8, мы будем применять пакет `fs` для операций с файловой системой:

```
library(fs)
data_dir <- path("pets_dataset")
dir_create(data_dir)

data_url <- path("http://www.robots.ox.ac.uk/~vgg/data/pets/data")
for (filename in c("images.tar.gz", "annotations.tar.gz")) {
  download.file(url = data_url / filename,
               destfile = data_dir / filename)
  untar(data_dir / filename, exdir = data_dir)
}
```

Входные изображения сохраняются в виде файлов JPG в папке `images/` (например, `images/Abyssinian_1.jpg`), а соответствующая маска сегментации сохраняется в виде файла PNG с тем же именем в папке `annotations/trimaps/` (например, `annotations/trimaps/Abyssinian_1.png`).

Давайте подготовим `data.frame` (технически `tibble` – табличку) со столбцами для наших путей к входным файлам, а также список соответствующих путей к файлам масок:

```
input_dir <- data_dir / "images"
target_dir <- data_dir / "annotations/trimaps/"

image_paths <- tibble::tibble(
  input = sort(dir_ls(input_dir, glob = "*.jpg")),
  target = sort(dir_ls(target_dir, glob = "*.png")))
```

Чтобы убедиться, что мы сопоставляем изображение с правильной целью, мы сортируем два списка. Векторы пути отсортируются одинаково, потому что пути исходных изображений и целей содержат одно и то же базовое имя файла. Затем, чтобы облегчить отслеживание путей и убедиться, что наши входные и целевые векторы остаются синхронизированными, мы объединяем их во фрейм данных с двумя столбцами (мы используем `tibble()` для создания `data.frame`):

```
tibble::glimpse(image_paths)
```

```
Rows: 7,390
```

```
Columns: 2
```

```
$ input <fs::path> "pets_dataset/images/Abyssinian_1.jpg", "pets_dataset/...
$ target <fs::path> "pets_dataset/annotations/trimaps/Abyssinian_1.png", "...
```

Как выглядит одно из входных изображений и его маска? Давайте посмотрим. Мы воспользуемся утилитами TensorFlow для чтения изображения и попутно поближе познакомимся с API. Сначала определим вспомогательную функцию, которая будет отображать тензор TensorFlow, содержащий изображение, с помощью функции `plot()` R:

По умолчанию вывод изображений без полей

```
display_image_tensor <- function(x, ..., max = 255,
                                plot_margins = c(0, 0, 0, 0)) {
```

```
  if(!is.null(plot_margins))
    par(mar = plot_margins)
```

```
  x %>%
```

```
  as.array() %>%
```

```
  drop() %>%
```

```
  as.raster(max = max) %>%
```

```
  plot(..., interpolate = FALSE)
```

```
}
```

interpolate = FALSE указывает графическому движку R рисовать пиксели с четкими краями, без смешивания или интерполяции цветов между пикселями

drop() удаляет оси с размером 1. Например, если x представляет собой изображение в градациях серого с одним цветовым каналом, он сожмет форму тензора с (height, width, 1) до (height, width)

Конвертация тензора в массив R

Конвертация массива R в растровый объект

В вызове `as.raster()` мы устанавливаем `max = 255`, потому что, как и в случае с MNIST, изображения кодируются как `uint8`, а 8-битные целые числа без знака могут кодировать значения только в диапазоне `[0, 255]`. Задавая `max = 255`, мы сообщаем графическому движку R отображать значения пикселей 255 как белые и 0 как черные и линейно интерполировать значения между ними в разные оттенки серого.

Теперь мы можем прочитать изображение в тензор и просмотреть его с помощью нашего вспомогательного метода `display_image_tensor()` (рис. 9.3):

```
library(tensorflow)
```

```
image_tensor <- image_paths$input[10] %>%
```

```
  tf$io$read_file() %>%
```

```
  tf$io$decode_jpeg()
```

```
str(image_tensor)
```

```
<tf.Tensor: shape=(448, 500, 3), dtype=uint8, numpy=...>
```

```
display_image_tensor(image_tensor)
```

Отображает входное изображение `Abyssinian_107.jpg`



Рис. 9.3 Пример изображения

Далее определим вспомогательную функцию для отображения целевого изображения. Целевое изображение также считывается как `uint8`, но на этот раз в тензоре целевого изображения находятся только значения (1, 2, 3). Чтобы отобразить его, мы вычитаем 1, чтобы метки находились в диапазоне от 0 до 2, а затем задаем `max = 2`, чтобы метки приобрели градации 0 (черный), 1 (серый) и 2 (белый).

Ниже показан код отображения цели и соответствующая цель (рис. 9.4):

```
display_target_tensor <- function(target)
  display_image_tensor(target - 1, max = 2)

target <- image_paths$target[10] %>%
  tf$io$read_file() %>%
  tf$io$decode_png()

str(target)
|<tf.Tensor: shape=(448, 500, 1), dtype=uint8, numpy=...>
display_target_tensor(target)
```



Рис. 9.4 Соответствующая целевая маска

Теперь загрузим наши входные данные и цели в два набора данных TF и разделим файлы на наборы для обучения и проверки. Поскольку набор данных очень мал, мы можем просто загрузить все в память:

```
library(tfdatasets)
tf_read_image <- function(path, format = "image", resize = NULL, ...) {
  img <- path %>%
    tf$io$read_file() %>%
    tf$io[[paste0("decode_", format)]](...)
  if (!is.null(resize))
```

Определяем вспомогательную функцию для чтения и изменения размера изображения с использованием операций TensorFlow

Находим `decode_image()`, `decode_jpeg()` или `decode_png()` в подмодуле `tf$io`

```

img <- img %>%
  tf$image$resize(as.integer(resize))
img
}

img_size <- c(200, 200)

tf_read_image_and_resize <- function(..., resize = img_size)
  tf_read_image(..., resize = resize)

```

Мы обязательно вызываем функцию модуля tf с целыми числами, используя as.integer()

Приводим изображения к размеру 200×200

Функция R, переданная в dataset\_map(), вызывается с символическими тензорами и должна возвращать символические тензоры. dataset\_map() получает здесь один аргумент – именованный список из двух тензоров скалярных строк, содержащий пути к файлам для входного и целевого изображений

```

make_dataset <- function(paths_df) {
  tensor_slices_dataset(paths_df) %>%
  dataset_map(function(path) {
    image <- path$input %>%
      tf_read_image_and_resize("jpeg", channels = 3L)
    target <- path$target %>%
      tf_read_image_and_resize("png", channels = 1L)
    target <- target - 1
    list(image, target)
  }) %>%
  dataset_cache() %>%
  dataset_shuffle(buffer_size = nrow(paths_df)) %>%
  dataset_batch(32)
}

```

Каждое входное изображение имеет три канала: значения RGB

Вычитаем 1, чтобы метки приняли значения 0, 1, 2

Каждое целевое изображение имеет один канал: целочисленные метки для каждого пикселя

Перетасуйте изображения, используя общее количество выборок в данных в качестве размера буфера. Мы обязательно вызываем shuffle после cache

Кеширование набора данных сохранит полный набор данных в памяти после первого запуска. Если на вашем компьютере недостаточно оперативной памяти, удалите этот вызов, и файлы изображений будут загружаться динамически по мере необходимости на протяжении всего обучения

```

num_val_samples <- 1000
val_idx <- sample.int(nrow(image_paths), num_val_samples)

```

Резервируем 1000 образцов для проверки

```

val_paths <- image_paths[val_idx, ]
train_paths <- image_paths[-val_idx, ]

```

Разделяем данные на обучающий и проверочный наборы

```

validation_dataset <- make_dataset(val_paths)
train_dataset <- make_dataset(train_paths)

```

Теперь можно определить нашу модель:

Определите локальные функции conv() и conv\_transpose(), чтобы мы могли избежать передачи одних и тех же аргументов при каждом вызове: padding = «same», активация = «relu»

```

get_model <- function(img_size, num_classes) {
  conv <- function(..., padding = "same", activation = "relu")
    layer_conv_2d(..., padding = padding, activation = activation)
  conv_transpose <- function(..., padding = "same", activation = "relu")

```

Мы везде используем padding = «same», чтобы избежать влияния отступов границ на размер карты объектов

```

layer_conv_2d_transpose(..., padding = padding, activation = activation)

input <- layer_input(shape = c(img_size, 3))
output <- input %>%
  layer_rescaling(scale = 1/255) %>%
  conv(64, 3, strides = 2) %>%
  conv(64, 3) %>%
  conv(128, 3, strides = 2) %>%
  conv(128, 3) %>%
  conv(256, 3, strides = 2) %>%
  conv(256, 3) %>%
  conv_transpose(256, 3) %>%
  conv_transpose(256, 3, strides = 2) %>%
  conv_transpose(128, 3) %>%
  conv_transpose(128, 3, strides = 2) %>%
  conv_transpose(64, 3) %>%
  conv_transpose(64, 3, strides = 2) %>%
  conv(num_classes, 3, activation = "softmax")

keras_model(input, output)
}

model <- get_model(img_size = img_size, num_classes = 3)

```

Не забудьте привести масштаб входных изображений к диапазону [0-1]

Мы заканчиваем модель трехвыводным слоем softmax для каждого пикселя, чтобы классифицировать каждый выходной пиксель в одну из наших трех категорий

Так выглядит краткая сводка модели:

```
model
```

```
Model: "model"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 200, 200, 3)]	0
rescaling (Rescaling)	(None, 200, 200, 3)	0
conv2d_6 (Conv2D)	(None, 100, 100, 64)	1792
conv2d_5 (Conv2D)	(None, 100, 100, 64)	36928
conv2d_4 (Conv2D)	(None, 50, 50, 128)	73856
conv2d_3 (Conv2D)	(None, 50, 50, 128)	147584
conv2d_2 (Conv2D)	(None, 25, 25, 256)	295168
conv2d_1 (Conv2D)	(None, 25, 25, 256)	590080
conv2d_transpose_5 (Conv2DTranspose)	(None, 25, 25, 256)	590080
conv2d_transpose_4 (Conv2DTranspose)	(None, 50, 50, 256)	590080
conv2d_transpose_3 (Conv2DTranspose)	(None, 50, 50, 128)	295040
conv2d_transpose_2 (Conv2DTranspose)	(None, 100, 100, 128)	147584
conv2d_transpose_1 (Conv2DTranspose)	(None, 100, 100, 64)	73792
conv2d_transpose (Conv2DTranspose)	(None, 200, 200, 64)	36928
conv2d (Conv2D)	(None, 200, 200, 3)	1731
Total params: 2,880,643		
Trainable params: 2,880,643		
Non-trainable params: 0		

Первая половина модели очень похожа на ту сеть, которую вы использовали для классификации изображений: набор слоев Conv2D

с постепенно увеличивающимися размерами фильтров. Мы в два раза уменьшаем каждое изображение выборки и проделываем это трижды, в результате чего получаем активацию размера (25, 25, 256). Назначение этой первой части модели заключается в кодировании изображений в более мелкие карты признаков, где каждая точка пространства (или пиксель) содержит информацию о большом пространственном фрагменте исходного изображения. Это можно рассматривать как своего рода сжатие.

Одно важное различие между первой половиной этой модели и моделями классификации, которые вы видели ранее, состоит в том, как мы выполняем понижение разрешения: в схемах классификации из предыдущей главы для понижения разрешения карт признаков мы использовали слои MaxPooling2D. В этой модели мы добавляем шаги свертки к каждому второму слою свертки (если вы не помните, что такое шаг свертки, вернитесь к разделу 8.1.1). Мы делаем это, потому что в случае сегментации изображения нас очень заботит *пространственное расположение* информации в изображении и нам нужно создавать попиксельные целевые маски в качестве выходных данных модели. Когда вы выполняете объединение по максимальному значению из соседних в окне  $2 \times 2$ , вы полностью уничтожаете информацию о местоположении в каждом окне объединения – вы возвращаете одно скалярное значение для каждого окна, не зная, какое из четырех местоположений в окне послужило источником значения. Следовательно, хотя слои MaxPooling2D хорошо работают для задач классификации, они плохо подходят для задачи сегментации. В то же время свертка с шагом лучше справляется с понижением размерности карт объектов, сохраняя при этом информацию о местоположении. В этой книге мы стараемся использовать свертку с шагом вместо объединения по ближайшему максимуму в любой модели, которая заботится о расположении признаков, например в генеративных моделях в главе 12.

Вторая половина модели представляет собой набор слоев Conv2DTranspose. Что это? Результат работы первой половины модели – это карта объектов формы (25, 25, 256), но нам нужно, чтобы наш окончательный результат имел ту же форму, что и целевые маски (200, 200, 3). Следовательно, нам нужно применить некое обратное преобразование, которое будет повышать размерность карт объектов, а не понижать ее. В этом и состоит назначение слоя Conv2DTranspose: вы можете рассматривать его как слой свертки, который учится повышать разрядность. Если у вас есть входные данные формы (100, 100, 64) и вы пропустите их через слой `layer_conv_2d(128, 3, strides = 2, padding = "same")`, то получите вывод формы (50, 50, 128). Если вы затем пропустите этот вывод через слой `layer_conv_2d_transpose(64, 3, strides = 2, padding = "same")`, то получите вывод (100, 100, 64), аналогичный исходной форме. Следовательно, после сжатия наших входных данных в карты объектов формы (25, 25, 256) с помощью стека слоев Conv2D мы можем просто применить соответствующую последовательность слоев Conv2DTranspose, чтобы вернуться к изображениям формы (200, 200, 3).

Теперь мы можем скомпилировать и обучить нашу модель:

```
model %>%
  compile(optimizer = "rmsprop",
          loss = "sparse_categorical_crossentropy")

callbacks <- list(
  callback_model_checkpoint("oxford_segmentation.keras",
                           save_best_only = TRUE))

history <- model %>% fit(
  train_dataset,
  epochs = 50,
  callbacks = callbacks,
  validation_data = validation_dataset
)
```

**ПРИМЕЧАНИЕ** Во время обучения вы можете увидеть предупреждение, например `Corrupt JPEG data: premature end of data segment` (Поврежденные данные JPEG: преждевременный конец сегмента данных). Набор данных изображения не идеален, но функции модуля `tf$io` могут корректно возобновлять свою работу после сбоя.

Построим графики потерь на обучающих и проверочных данных (рис. 9.5):

```
plot(history)
```

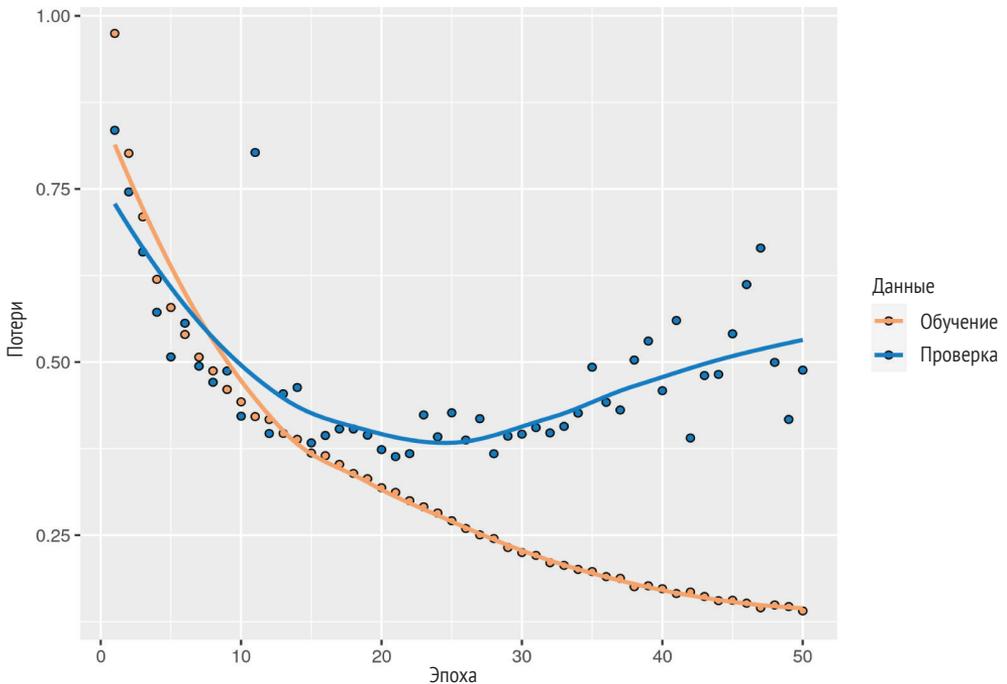


Рис. 9.5 Кривые потерь при обучении и проверке

Можно заметить, что переобучение начинается на полпути, примерно в эпоху 25. Загрузим нашу лучшую модель, сохраненную в соответствии с потерями при проверке, и посмотрим, как она работает при прогнозировании маски сегментации (рис. 9.6):

```

model <- load_model_tf("oxford_segmentation.keras")

test_image <- val_paths$input[309] %>%
  tf_read_image_and_resize("jpeg", channels = 3L)

predicted_mask_probs <-
  model(test_image[tf$newaxis, , ])
predicted_mask <-
  tf$argmax(predicted_mask_probs, axis = -1L)

predicted_target <- predicted_mask + 1

par(mfrow = c(1, 2))
display_image_tensor(test_image)
display_target_tensor(predicted_target)

```

tf\$newaxis добавляет измерение пакетов, потому что наша модель ожидает пакеты изображений. model() возвращает тензор с shape=(1, 200, 200, 3), dtype=float32

tf\$argmax() похож на which.max() в R. Основное отличие состоит в том, что tf\$argmax() возвращает значения, начинающиеся с 0. Базовый эквивалент R для tf\$argmax(x, axis = -1L) – это apply(x, c(1, 2, 3), which.max) – 1L

Предсказанная маска представляет собой тензор с shape=(1, 200, 200), dtype=int64

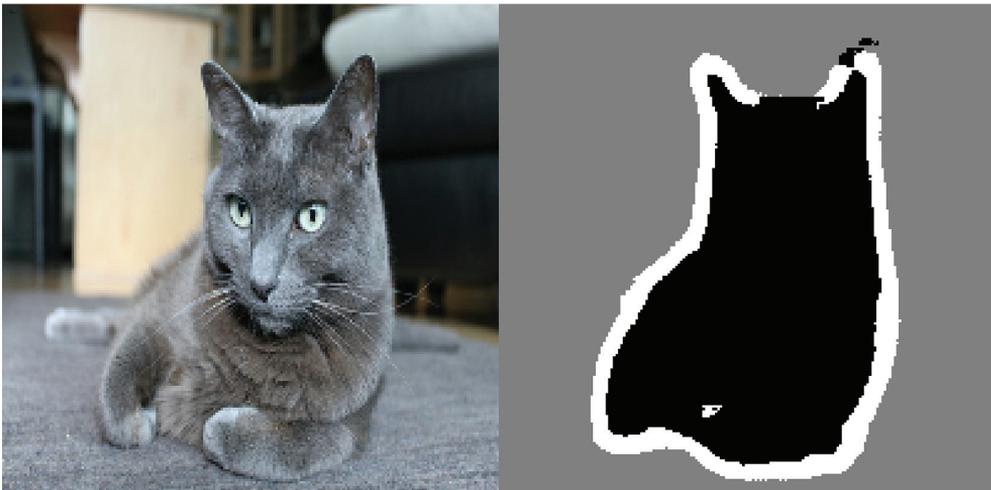


Рис. 9.6 Тестовое изображение и его предсказанная маска сегментации

В предсказанной маске есть пара небольших артефактов. Тем не менее наша модель работает достаточно хорошо.

К настоящему времени вы изучили основы того, как выполнять классификацию и сегментацию изображений: вы уже можете многого добиться с тем, что знаете. Однако сверточные сети, которые опытные инженеры разрабатывают для решения реальных проблем, не так просты, как те, которые мы до сих пор использовали в наших

примерах. Вам по-прежнему не хватает знаний и навыков, которые позволяют экспертам принимать быстрые и точные решения о том, как построить современные модели. Чтобы заполнить этот пробел, вам необходимо глубже изучить стандартные архитектуры. Мы займемся этим в следующем разделе.

## 9.3 Современные стандартные архитектуры сверточных сетей

*Архитектура модели* – это сумма решений, которые легли в ее основу: какие слои использовать, как их настраивать и в каком порядке их соединять. Сочетание этих решений определяет *пространство гипотез* вашей модели, то есть пространство возможных функций, которые может искать градиентный спуск, параметризованное весами модели. Подобно конструированию признаков, хорошее пространство гипотез кодирует имеющиеся у вас *предварительные* (или *априорные*) знания о проблеме и ее решении. Например, использование слоев свертки означает априорное знание о том, что соответствующие закономерности, присутствующие во входных изображениях, инвариантны к трансляции. Чтобы эффективно обучить модель на данных, вам сначала нужно сделать обоснованные предположения о том, что вы ищете.

Граница между успехом и неудачей часто определяется архитектурой модели. Если вы неправильно выберете архитектуру, ваша модель может застрять на одном месте с неоптимальными метриками, и никакие обучающие данные ее не спасут. И наоборот, хорошая архитектура модели ускорит обучение и позволит вашей модели эффективно использовать доступные обучающие данные, уменьшая потребность в больших наборах. Хорошая архитектура модели – это та, которая *уменьшает размер пространства поиска или иным образом упрощает сходимость к оптимальной точке пространства поиска*. Точно так же, как конструирование признаков и подготовка данных, выбор архитектуры модели призван упростить решение задачи градиентным спуском. Не забывайте, что градиентный спуск – довольно глупый процесс поиска, которому нужно предоставить максимально возможную помощь.

Разработка архитектуры модели – это больше искусство, чем наука. Опытные специалисты по машинному обучению могут интуитивно построить эффективную модель с первой попытки, в то время как новичкам бывает трудно создать модель, которая хоть как-то обучается. Ключевое слово здесь – *интуитивно*. Никто не может дать вам четкие пояснения, что работает, а что нет. Эксперты полагаются на сопоставление с похожими решениями, способность, которую они приобретают благодаря обширному практическому опыту. Разумеется, эта книга тоже способствует развитию интуиции.

Однако дело не только в интуиции – как и в любой технической дисциплине, в глубоком обучении есть свои лучшие приемы и тонкости, которые следует знать.

В следующих разделах мы рассмотрим несколько важных передовых методов построения архитектур сверточных сетей: в частности, *остаточные связи* (residual connections), *пакетную нормализацию* (batch normalization) и *разделяемые свертки* (separable convolutions). Овладев этими методами, вы сможете создавать высокоэффективные модели для работы с изображениями. В этой книге мы применим их к задаче классификации изображений кошек и собак.

Начнем издадалека – с формулы «модульность – иерархия – повторное использование» (modularity-hierarchy-reuse, MHR) для системной архитектуры.

### 9.3.1 Модульность, иерархия и повторное использование

Если вы хотите упростить сложную систему, попробуйте применить универсальный рецепт: сначала организуйте элементы своей системы в *модули*, затем организуйте модули в *иерархию* и начните *повторно использовать* одни и те же модули в нескольких местах по мере необходимости («повторное использование» в данном контексте служит аналогом *абстракции*). Это знаменитая формула MHR, лежащая в основе системной архитектуры практически во всех областях, где используется термин «архитектура». В соответствии с этой формулой организована любая система значимой сложности, будь то кафедральный собор, ваше собственное тело, ВМС США или кодовая база Keras (рис. 9.7).

Если вам довелось профессионально заниматься программированием, вы уже хорошо знакомы с этими принципами: эффективная кодовая база – это модульная и иерархическая структура, в которой вы не реализуете одну и ту же вещь дважды, а вместо этого полагаетесь на многократно используемые классы и функции. Если вы структурируете свой код, следуя этим принципам, можно сказать, что вы занимаетесь «архитектурой программного обеспечения».

Само по себе глубокое обучение – это просто применение формулы MHR к непрерывной оптимизации с помощью градиентного спуска: вы берете классический метод оптимизации (градиентный спуск по непрерывному функциональному пространству) и структурируете пространство поиска в модули (слои), организованные в глубокую иерархию (часто просто стек, простейший вид иерархии), где вы повторно используете все, что можете (например, свертки – это повторное использование 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. Вы могли заметить, что все популярные архитектуры сверточных сетей состоят не только из

слоев, но и из повторяющихся групп слоев (называемых «блоками» или «модулями»). Например, популярная архитектура VGG16, которую мы использовали в предыдущей главе, состоит из повторяющихся блоков «свертка – свертка – максимум» (рис. 9.8).

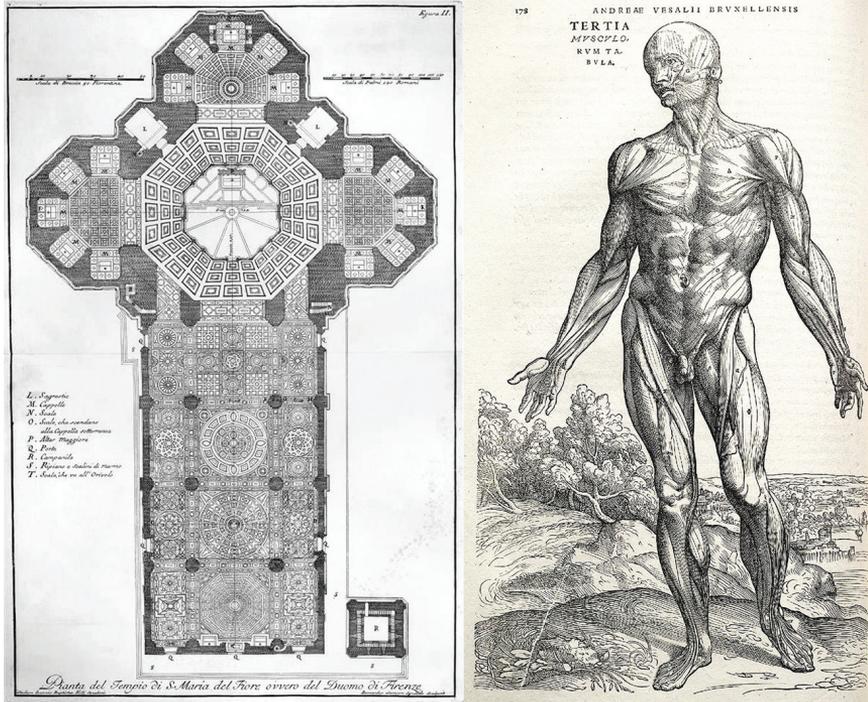


Рис. 9.7 Сложные системы имеют иерархическую структуру и организованы в отдельные модули, которые многократно используются повторно (например, ваши четыре конечности, которые представляют собой варианты одного и того же рычага, или ваши 20 пальцев)

Кроме того, большинство сверточных сетей имеют пирамидальную структуру (*иерархию признаков*). Вспомните, например, прогрессию количества сверточных фильтров, которые мы использовали в первой сверточной сети, созданной нами в предыдущей главе: 32, 64, 128. Количество фильтров увеличивается с глубиной слоя, тогда как размер карт признаков соответственно уменьшается. Вы заметите ту же закономерность в блоках модели VGG16 (рис. 9.8).

Более глубокие иерархии хороши хотя бы потому, что они поощряют повторное использование признаков и, следовательно, абстракцию. В целом глубокая стопка узких слоев работает лучше, чем неглубокая стопка широких слоев. Однако из-за проблемы исчезающих градиентов существует ограничение на максимальную глубину слоев. Это приводит нас к нашему первому базовому шаблону архитектуры: остаточным связям.

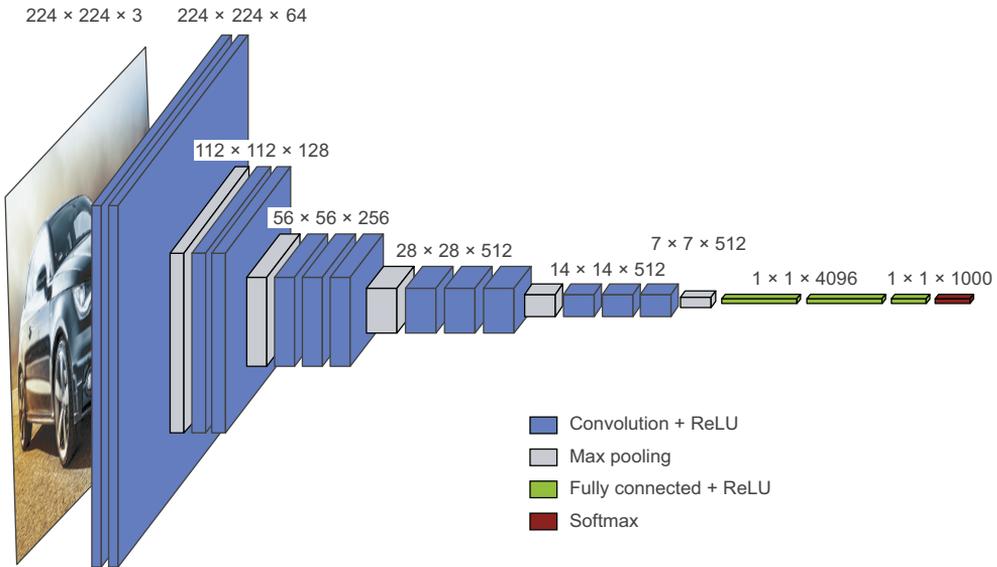


Рис. 9.8 Архитектура модели VGG16. Обратите внимание на повторяющиеся блоки слоев и пирамидальную структуру карт признаков

### О важности изучения абляции в исследованиях глубокого обучения

Архитектуры глубокого обучения зачастую скорее *эволюционируют*, чем создаются с нуля – они получаются путем многочисленных проб и ошибок с последующим отбором наиболее успешных вариантов. Как и в случае с живыми организмами, если вы возьмете любую сложную экспериментальную модель глубокого обучения, скорее всего, вы сможете удалить несколько модулей (или заменить некоторые обученные признаки случайными) без заметной потери качества прогнозов.

Ситуацию усугубляют сомнительные стимулы, с которыми сталкиваются исследователи глубокого обучения: делая систему более сложной, чем необходимо, они могут сделать ее более интересной или более новой и, таким образом, убедить рецензентов в необходимости публикации статьи. Если вы прочитаете много статей по глубокому обучению, вы заметите, что их часто подгоняют для благоприятных отзывов рецензентов как по стилю, так и по содержанию, что существенно снижает ясность объяснений и надежность результатов. Например, математику в работах по глубокому обучению редко применяют для четкой формализации концепций или получения неочевидных результатов – скорее, ее используют как символ собственной серьезности, как дорогой костюм на коммивояжере.

Целью исследования должна быть не просто публикация очередной статьи, а создание надежных знаний. Важно отметить, что понимание *причинно-следственной связи* в вашей системе – самый простой способ получить надежные знания. И есть очень простой способ изучить при-

чинно-следственную связь: *абляционное исследование*. Это исследование заключается в систематических попытках удаления (*абляции*) части системы, в ее упрощении, чтобы определить, что именно определяет ее эффективность. Если вы обнаружите, что  $X + Y + Z$  дает хорошие результаты, попробуйте также варианты  $X + Y$ ,  $X + Z$  и  $Y + Z$  и посмотрите, что произойдет.

Если вы займетесь исследованиями в области глубокого обучения, избавляйтесь от шума в объекте исследования – проводите абляцию своих моделей. Всегда задавайте себе вопросы: «Можно ли предложить более простое объяснение? Действительно ли необходима эта дополнительная сложность? Почему?»

### 9.3.2 Остаточные связи

Вы, вероятно, знаете об игре «Испорченный телефон», где первый игрок шепчет исходное сообщение на ухо второму игроку, тот шепчет его на ухо третьему игроку и т. д. Сообщение, полученное последним игроком, в конечном итоге мало похоже на исходную версию. Это забавная метафора совокупных ошибок, возникающих при последовательной передаче по зашумленному каналу.

Как оказалось, обратное распространение в последовательной модели глубокого обучения очень похоже на игру «Испорченный телефон». У вас есть цепочка функций наподобие такой:

$$y = f_4(f_3(f_2(f_1(x))))$$

Суть игры состоит в том, чтобы настроить параметры каждой функции в цепочке на основе ошибки, обнаруженной на выходе  $f_4$  (потеря модели). Чтобы настроить  $f_1$ , вам нужно будет передать информацию об ошибках через  $f_2$ ,  $f_3$  и  $f_4$ . Однако каждая последующая функция в цепочке вносит некоторый шум. Если ваша цепочка функций слишком длинная, этот шум начинает подавлять информацию о градиенте, и обратное распространение перестает работать. Ваша модель вообще не будет обучаться. Это проблема *исчезающих градиентов*.

Решение простое: сделайте каждую функцию в цепочке неразрушающей, чтобы сохранить бесшумную версию информации, поступившей на предыдущий ввод. Самый простой способ реализовать это – использовать *остаточные связи* (residual connection, иногда их называют *обходными связями*).

Для этого сложите входной сигнал слоя или блока с его выходным сигналом (рис. 9.9). Остаточная связь действует как информационный обходной путь мимо деструктивных или зашумленных блоков (например, блоки, содержащие слои ReLU-активации или прореживания), позволяя незашумленной информации о градиенте ошибок из ранних слоев распространяться по глубокой сети. Этот метод был

представлен в 2015 году в семействе моделей ResNet (разработанных Хе и др. в Microsoft)<sup>1</sup>.

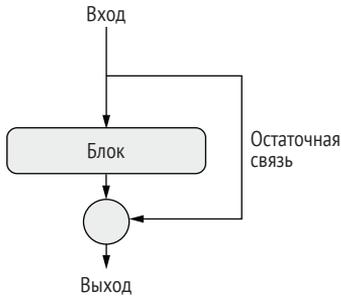


Рис. 9.9 Остаточная связь, направленная в обход блока обработки

На практике можно реализовать остаточную связь, как показано в листинге 9.1.

#### Листинг 9.1 Реализация остаточной связи (псевдокод)

```
x <- ... ← Некоторый входной тензор
residual <- x
x <- block(x) ←
x <- layer_add(c(x, residual)) ←
```

Этот блок вычислений потенциально может вносить в данные искажения или шум, но это не проблема

Сохраняем указатель на исходный ввод. Он называется остатком

Прибавляем исходный ввод к выводу слоя: таким образом, окончательный вывод всегда будет содержать полную информацию об исходном вводе

Разумеется, операция сложения ввода и вывода блока подразумевает, что они должны иметь одинаковую форму. Однако это условие не выполняется, если блок содержит сверточные слои с увеличенным количеством фильтров или слой с объединением по максимальному значению из соседних. В таких случаях используйте слой `layer_conv_2d()`  $1 \times 1$  без активации, чтобы линейно проецировать невязку на желаемую выходную форму (листинг 9.2). Обычно вы будете использовать параметр `padding = "same"` в слоях свертки в целевом блоке, чтобы избежать пространственного понижения дискретизации из-за заполнения, и вы должны применять шаги в остаточной проекции, чтобы уравнивать любое понижение разрядности, вызванное слоем объединения по максимальному значению (листинг 9.3).

#### Листинг 9.2 Остаточный блок, в котором изменяется количество фильтров

```
inputs <- layer_input(shape = c(32, 32, 3))
x <- inputs %>% layer_conv_2d(32, 3, activation = "relu")
```

<sup>1</sup> Kaiming He et al., *Deep Residual Learning for Image Recognition*, Conference on Computer Vision and Pattern Recognition (2015), <https://arxiv.org/abs/1512.03385>.

Откладываем остаток

Это слой, вокруг которого мы создаем остаточное соединение: он увеличивает количество выходных фильтров с 32 до 64. Обратите внимание, что мы используем padding = «same», чтобы избежать понижения частоты дискретизации из-за заполнения

```
residual <- x
x <- x %>% layer_conv_2d(64, 3, activation = "relu", padding = "same")
residual <- residual %>% layer_conv_2d(64, 1)
x <- layer_add(c(x, residual))
```

Теперь выход блока и остаток имеют одинаковую форму и можно выполнить операцию сложения

У остатка было только 32 фильтра, поэтому мы используем слой layer\_conv\_2d 1x1, чтобы спроецировать его в правильную форму

### Листинг 9.3 Случай, когда целевой блок включает слой объединения по максимальному соседнему

Откладываем остаток

Это блок из двух слоев, в обход которого мы создаем остаточную связь: он включает слой объединения по максимальному значению 2x2. Обратите внимание, что мы используем padding = «same» как в слое свертки, так и в слое объединения, чтобы избежать понижения размерности

```
inputs <- layer_input(shape = c(32, 32, 3))
x <- inputs %>% layer_conv_2d(32, 3, activation = "relu")
residual <- x
x <- x %>%
  layer_conv_2d(64, 3, activation = "relu", padding = "same") %>%
  layer_max_pooling_2d(2, padding = "same") %>%
  residual <- residual %>%
  layer_conv_2d(64, 1, strides = 2)
x <- layer_add(list(x, residual))
```

Теперь вывод блока и данные остаточной связи имеют одинаковую форму и их можно складывать

Мы используем stride = 2 в остаточной проекции, чтобы соответствовать уменьшению размерности, созданному слоем объединения

Для наглядности рассмотрим пример простой сверточной сети, структурированной в виде последовательности блоков, каждый из которых состоит из двух слоев свертки и одного дополнительно-го слоя объединения по максимальному соседнему, с остаточной связью в обход каждого блока:

Вспомогательная функция для применения блока свертки с остаточной связью с возможностью добавления объединения по максимальному из соседних

```
inputs <- layer_input(shape = c(32, 32, 3))
x <- layer_rescaling(inputs, scale = 1/255)

residual_block <- function(x, filters,
                           pooling = FALSE) {
  residual <- x
  x <- x %>%
    layer_conv_2d(filters, 3, activation = "relu", padding = "same") %>%
    layer_conv_2d(filters, 3, activation = "relu", padding = "same")

  if (pooling) {
    x <- x %>% layer_max_pooling_2d(pool_size = 2, padding = "same")
```

Если мы используем объединение, то добавляем пошаговую свертку, чтобы спроецировать остаток в ожидаемую форму

```

                                Если мы не используем объединение, то проецируем остаток,
                                только если количество каналов изменилось
    residual <- residual %>% layer_conv_2d(filters, 1, strides = 2)
  } else if (filters != dim(residual)[4]) { ←
    residual <- residual %>% layer_conv_2d(filters, 1)
  }

  layer_add(list(x, residual))
}

outputs <- x %>%
  residual_block(filters = 32, pooling = TRUE) %>%
  residual_block(filters = 64, pooling = TRUE) %>% ←
  residual_block(filters = 128, pooling = FALSE) %>% ←
  layer_global_average_pooling_2d() %>%
  layer_dense(units = 1, activation = "sigmoid")

model <- keras_model(inputs = inputs, outputs = outputs)

```

Первый блок

Второй блок; обратите внимание на увеличение количества фильтров в каждом блоке

Последний блок не нуждается в слое объединения по максимальному соседнему, потому что сразу после него мы применим глобальное объединение средних значений

Так выглядит сводка модели, которая у нас получилась:

```
model
```

```
Model: "model_1"
```

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 32, 32, 3)]	0	[]
rescaling_1 (Rescaling)	(None, 32, 32, 3)	0	['input_2[0][0]']
conv2d_8 (Conv2D)	(None, 32, 32, 32)	896	['rescaling_1[0][0]']
conv2d_7 (Conv2D)	(None, 32, 32, 32)	9248	['conv2d_8[0][0]']
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0	['conv2d_7[0][0]']
conv2d_9 (Conv2D)	(None, 16, 16, 32)	128	['rescaling_1[0][0]']
add (Add)	(None, 16, 16, 32)	0	['max_pooling2d[0][0]', 'conv2d_9[0][0]']
conv2d_11 (Conv2D)	(None, 16, 16, 64)	18496	['add[0][0]']
conv2d_10 (Conv2D)	(None, 16, 16, 64)	36928	['conv2d_11[0][0]']
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0	['conv2d_10[0][0]']
conv2d_12 (Conv2D)	(None, 8, 8, 64)	2112	['add[0][0]']
add_1 (Add)	(None, 8, 8, 64)	0	['max_pooling2d_1[0][0]', 'conv2d_12[0][0]']
conv2d_14 (Conv2D)	(None, 8, 8, 128)	73856	['add_1[0][0]']
conv2d_13 (Conv2D)	(None, 8, 8, 128)	147584	['conv2d_14[0][0]']
conv2d_15 (Conv2D)	(None, 8, 8, 128)	8320	['add_1[0][0]']
add_2 (Add)	(None, 8, 8, 128)	0	['conv2d_13[0][0]', 'conv2d_15[0][0]']
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0	['add_2[0][0]']

```
dense (Dense)                (None, 1)                129    ['global_average_
                                     ➔ pooling2d[0][0]']
=====
Total params: 297,697
Trainable params: 297,697
Non-trainable params: 0
```

С остаточными связями вы можете строить сети произвольной глубины, не беспокоясь об исчезновении градиентов.

Далее мы рассмотрим следующий базовый шаблон архитектуры сверточных сетей: *пакетной нормализации*.

### 9.3.3 Пакетная нормализация

*Нормализация* – это обширная категория методов, которые предназначены для того, чтобы сделать различные образцы, доступные модели машинного обучения, более похожими друг на друга, что помогает модели хорошо учиться и обобщать новые данные. Наиболее распространенная форма нормализации данных – та, которую вы уже видели несколько раз в этой книге: центрирование данных к нулю путем вычитания их среднего значения и придание данным единичного стандартного отклонения путем деления на их стандартное отклонение. По сути, нормализация задает предположение, что данные следуют нормальному (или гауссову) распределению, и гарантирует, что это распределение центрировано и масштабировано до единичной дисперсии:

```
normalize_data <- apply(data, <axis>, function(x) (x - mean(x)) / sd(x))
```

В предыдущих примерах в этой книге мы нормализовали данные перед их вводом в модели. Но есть смысл выполнять нормализацию после каждого преобразования, выполняемого сетью: даже если данные, поступающие в сеть из слоев Dense или Conv2D, имеют нулевое среднее значение и единичную дисперсию, нет никаких оснований ожидать априори, что это будет справедливо для выходных данных. Может ли нормализация промежуточных активаций улучшить качество модели?

Именно это и делает *пакетная нормализация* (batch normalization). Это особый тип слоя (layer\_batch\_normalization() в Keras), представленный в 2015 году Иоффе и Сегеди<sup>1</sup>; он может адаптивно нормализовать данные даже при изменении среднего значения и дисперсии во время обучения. Во время обучения он использует среднее значение и дисперсию текущего пакета данных для нор-

<sup>1</sup> Sergey Ioffe, Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, Proceedings of the 32nd International Conference on Machine Learning (2015), <https://arxiv.org/abs/1502.03167>.

мализации выборок, а во время логического вывода (когда достаточно большой пакет репрезентативных данных может быть недоступен) он использует экспоненциальное скользящее среднее для среднего значения пакета и дисперсии данных, полученные во время обучения.

Хотя в исходной статье сказано, что нормализация пакетов работает за счет «уменьшения внутреннего ковариатного сдвига», никто точно не знает, почему нормализация пакетов помогает на самом деле. Существуют различные гипотезы, но ни в одной из них нет полной уверенности. Вы обнаружите, что в глубоком обучении такая неопределенность встречается очень часто – это не точная наука, а набор постоянно меняющихся, эмпирически полученных передовых методик, связанных между собой ненадежными предположениями. Иногда вам будет казаться, что книга, которую вы держите в руках, подробно рассказывает, *как* что-то делать, но слишком мало говорит о том, *почему* это работает, – дело в том, что про многие вещи мы знаем, «как», но не знаем, «почему». Всякий раз, когда доступно надежное объяснение, я обязательно упоминаю его. Пакетная нормализация не относится к таким случаям.

На практике основной эффект пакетной нормализации, по-видимому, заключается в том, что она способствует распространению градиента – примерно так же, как остаточные соединения – и, таким образом, позволяет создавать более глубокие сети. Некоторые очень глубокие сети удается обучить, только если они включают несколько слоев пакетной нормализации. Например, пакетная нормализация широко используется во многих передовых архитектурах сверточных сетей, поставляемых вместе с Keras, таких как ResNet50, EfficientNet и Xception.

Слой `layer_batch_normalization()` можно применять после любого слоя – `layer_dense()`, `layer_conv_2d()` и т. д.:

```
x <- ... ← | Например, layer_input(), keras_model_sequential()
              | или вывод другого слоя
x <- x %>%
  layer_conv_2d(32, 3, use_bias = FALSE) %>% ←
  layer_batch_normalization()
                                             | Поскольку вывод layer_conv_2d() нормализован,
                                             | слою не нужен собственный вектор смещения
```

И `layer_dense()`, и `layer_conv_2d()` используют *вектор смещения*, обучаемую переменную, целью которой является сделать слой *аффинным*, а не просто линейным. Например, `layer_conv_2d()`, упрощенно говоря, возвращает  $y = \text{conv}(x, \text{kernel}) + \text{bias}$ , а `layer_dense()` возвращает  $y = \text{dot}(x, \text{kernel}) + \text{bias}$ . Поскольку шаг нормализации позаботится о центрировании вывода слоя к нулю, при использовании `layer_batch_normalization()` вектор смещения больше не нужен, и слой можно создать без него с помощью параметра `use_bias = FALSE`. Это делает слой немного проще.

Нужно отметить, что обычно я рекомендую размещать активацию предыдущего слоя после слоя пакетной нормализации (хотя это все еще является предметом дискуссий). Следовательно, вместо кода, показанного в листинге 9.4, вы должны применять код из листинга 9.5.

#### Листинг 9.4 Как *не* следует использовать пакетную нормализацию

```
x %>%
  layer_conv_2d(32, 3, activation = "relu") %>%
  layer_batch_normalization()
```

#### Листинг 9.5 Как правильно использовать пакетную нормализацию: активация выполняется последней

```
x %>%
  layer_conv_2d(32, 3, use_bias = FALSE) %>%
  layer_batch_normalization() %>%
  layer_activation("relu")
```

Обратите внимание на отсутствие активации в этом месте

Мы размещаем активацию после `layer_batch_normalization()`

Интуитивно понятная причина такого подхода заключается в том, что при пакетной нормализации ваши входные данные будут центрированы к нулю, тогда как ваша активация `relu` использует ноль в качестве опорной точки для сохранения или удаления активированных каналов: выполнение нормализации перед активацией максимизирует использование `relu`. Тем не менее эта современная практика упорядочения слоев не совсем критична, поэтому если вы выполняете свертку, затем активацию, а потом пакетную нормализацию, ваша модель все равно будет обучаться, и результат не обязательно будет хуже, чем в моем варианте.

### О пакетной нормализации и дообучении модели

Пакетная нормализация имеет много особенностей. Одна из основных связана с дообучением: при дообучении модели, включающей слои `BatchNormalization`, я рекомендую оставить эти слои замороженными (вызвать метод `freeze_weights()`, чтобы установить для них атрибут `trainable = FALSE`). В противном случае они продолжат обновлять свое внутреннее среднее значение и дисперсию, что может помешать очень небольшим обновлениям, применяемым к окружающим слоям `Conv2D`:

```
batch_norm_layer_s3_classname <- class(layer_batch_normalization())[1]
batch_norm_layer_s3_classname
```

```
[1] "keras.layers.normalization.batch_normalization."
➔ BatchNormalization"
```

```
is_batch_norm_layer <- function(x)
  inherits(x, batch_norm_layer_s3_classname)
```

```

model <- application_efficientnet_b0()
for(layer in model$layers)
  if(is_batch_norm_layer(layer))
    layer$trainable <- FALSE

```

Пример того, как установить `trainable <- FALSE`, чтобы заморозить только слои BatchNormalization. Примечание: вы также можете вызвать метод `freeze_weights(model, which = is_batch_norm_layer)` для достижения того же результата

Теперь рассмотрим последний шаблон архитектуры в нашей серии: разделяемые по глубине свертки.

### 9.3.4 Разделяемые по глубине свертки

Представьте, что в вашем распоряжении есть слой, который можно использовать в качестве замены для `layer_conv_2d()`. Этот слой делает вашу модель компактнее (уменьшает количество обучаемых весовых параметров) и легче в плане вычислений (меньше операций с плавающей запятой), а также повышает точность на несколько процентов. Мы говорим про слой *свертки с разделением по глубине* (`layer_separable_conv_2d()` в Keras). Этот слой выполняет пространственную свертку на каждом входном канале независимо перед микшированием выходных каналов посредством *точечной свертки* (свертка  $1 \times 1$ ), как показано на рис. 9.10.

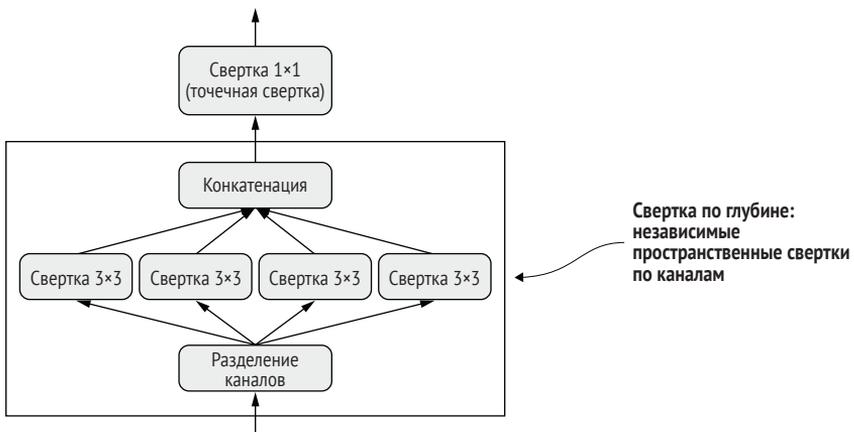


Рис. 9.10 Разделяемая по глубине свертка: за глубиной сверткой следует точечная свертка

Это эквивалентно разделению изучения пространственных признаков и признаков канала. Во многом так же, как обычная пространственная свертка основана на предположении, что паттерны в изображениях не связаны с конкретными местоположениями,

свертка с разделением по глубине основана на предположении, что *пространственные местоположения* в промежуточных активациях *сильно коррелированы*, но разные каналы *строго независимы*. Поскольку это предположение обычно верно для представлений изображений, изученных глубокими нейронными сетями, оно служит полезным априорным показателем, который помогает модели более эффективно использовать свои обучающие данные. Модель с более сильными априорными данными о структуре информации, которую ей придется обрабатывать, является лучшей моделью – до тех пор, пока априорные значения точны.

Разделяемая по глубине свертка требует значительно меньшего количества параметров и меньше вычислений по сравнению с обычной сверткой, имея при этом сравнимую репрезентативную мощность, что позволяет построить более легкие модели, которые сходятся быстрее и менее склонны к переобучению. Эти преимущества становятся особенно важными, когда вы обучаете небольшие модели с нуля на ограниченных данных.

Если говорить о крупномасштабных моделях, разделяемые по глубине свертки являются основой архитектуры Xception, высокопроизводительной сети, поставляемой вместе с Keras. Вы можете больше узнать о теоретическом обосновании глубоких отделимых сверток и Xception в статье «Xception: Deep Learning with Depthwise Separable Convolutions»<sup>1</sup>.

### Совместная эволюция аппаратного/программного обеспечения и алгоритмов

Рассмотрим обычную операцию свертки с окном  $3 \times 3$ , 64 входными и 64 выходными каналами. Она использует  $3 * 3 * 64 * 64 = 36\,864$  обучаемых параметра. Когда мы применяем операцию свертки к изображению, необходимо выполнить ряд операций с плавающей запятой, количество которых пропорционально значению этого параметра. Теперь рассмотрим эквивалентную свертку с разделением по глубине: она использует всего  $3 * 3 * 64 + 64 * 64 = 4672$  обучаемых параметра и пропорционально меньше операций с плавающей запятой. Это повышение эффективности становится все более заметным по мере увеличения количества фильтров или размера окон свертки.

В результате вы ожидаете, что свертки, разделяемые по глубине, будут работать намного быстрее, верно? Не спешите с выводами. Это было бы верно, если бы в том и другом случае вы писали простые реализации этих алгоритмов на CUDA или C – действительно, вы увидите значительное ускорение при использовании обычного CPU, где базовая реализация яв-

<sup>1</sup> François Chollet, *Xception: Deep Learning with Depthwise Separable Convolutions*, Conference on Computer Vision and Pattern Recognition (2017), <https://arxiv.org/abs/1610.02357>.

ляется распараллеленным выполнением кода C. Но на практике вы, скорее всего, используете графический процессор, и то, что на нем выполняется, далеко не «простая» реализация CUDA: это ядро cuDNN, фрагмент кода, который был невероятно оптимизирован вплоть до каждой машинной инструкции. Конечно, затраченные на оптимизацию усилия давно окупились, так как свертки cuDNN на оборудовании NVIDIA отвечают за экзафлопсы вычислений каждый день. Но побочным эффектом такой экстремальной оптимизации является то, что у альтернативных подходов почти нет шансов составить конкуренцию по производительности – даже у архитектур, которые имеют значительные внутренние преимущества, такие как свертки с разделением по глубине.

Несмотря на неоднократные обращения пользователей к NVIDIA, свертки с разделением по глубине не получили такого же уровня оптимизации программного и аппаратного обеспечения, что и обычные свертки, и в результате у них остается примерно такое же быстродействие, как у обычных сверток, даже несмотря на то что они используют квадратично меньше параметров и операций с плавающей запятой. Стоит заметить, однако, что использование сверток с разделением по глубине остается хорошей идеей, даже если это не приводит к ускорению: меньшее количество параметров означает, что модель меньше подвержена риску переобучения, а предположение о том, что каналы должны быть некоррелированы, приводит к более быстрой сходимости модели и более надежным представлениям.

Небольшое неудобство в данном случае может стать непреодолимой стеной в других ситуациях: поскольку вся программно-аппаратная экосистема глубокого обучения была жестко оптимизирована под очень специфический набор алгоритмов (в частности, сверточные сети, обученные методом обратного распространения), чрезвычайно высока цена отказа от проторенного пути.

Если вы начнете экспериментировать с альтернативными алгоритмами, такими как оптимизация без градиента или импульсные нейронные сети, первые несколько параллельных реализаций на C++ или CUDA, которые вы придумаете, будут на несколько порядков медленнее, чем старые добрые сверточные сети, независимо от того, насколько умны и эффективны ваши идеи. Вам будет сложно убедить других исследователей принять ваш метод, даже если потенциально он намного лучше.

Можно сказать, что современное глубокое обучение является продуктом процесса совместной эволюции аппаратного, программного обеспечения и алгоритмов: доступность графических процессоров NVIDIA и CUDA привела к успеху сверточных сетей, обученных обратным распространением, что побудило NVIDIA оптимизировать свое оборудование и программное обеспечение для этих алгоритмов, что, в свою очередь, привело к укреплению исследовательского сообщества, стоящего за этими методами. На данный момент для формирования другого пути потребуются многолетняя реорганизация всей экосистемы.

### 9.3.5 Применим знания на практике: мини-модель, подобная Xception

Вспомним базовые принципы архитектуры сверточных сетей, которые вы уже изучили:

- ваша модель должна быть организована в виде повторяющихся блоков слоев, обычно состоящих из нескольких слоев свертки и слоя с объединением по максимальному соседнему;
- количество фильтров в ваших слоях должно увеличиваться по мере уменьшения размера карт пространственных признаков;
- лучше глубокая сеть с узкими слоями, чем мелкая сеть с широкими слоями;
- добавление остаточных связей в обход блоков слоев помогает обучать более глубокие сети;
- часто бывает полезно ввести слои пакетной нормализации после слоев свертки;
- бывает полезно заменить слои `layer_conv_2d()` на `layer_separable_conv_2d()`, которые более эффективны с точки зрения параметров.

Давайте объединим эти идеи в одной модели. Ее архитектура будет напоминать уменьшенную версию Xception, и мы применим ее к задаче классификации «собака или кошка» из предыдущей главы. Для загрузки данных и обучения модели мы просто повторно используем структуру, которую применяли в разделе 8.2.5, но заменим определение модели следующей сверточной сетью:

```
data_augmentation <- keras_model_sequential() %>%
  layer_random_flip("horizontal") %>%
  layer_random_rotation(0.1) %>%
  layer_random_zoom(0.2)
```

Мы используем ту же конфигурацию расширения данных, что и раньше

Не забудьте масштабировать входные данные

```
inputs <- layer_input(shape = c(180, 180, 3))
x <- inputs %>%
  data_augmentation() %>%
  layer_rescaling(scale = 1 / 255)
```

```
x <- x %>%
  layer_conv_2d(32, 5, use_bias = FALSE)
```

```
for (size in c(32, 64, 128, 256, 512)) {
  residual <- x
```

```
  x <- x %>%
    layer_batch_normalization() %>%
    layer_activation("relu") %>%
```

Обратите внимание, что предположение, лежащее в основе свертки с разделением, о том, что «каналы признаков в значительной степени независимы», не выполняется для изображений RGB! Каналы красного, зеленого и синего цветов на самом деле сильно коррелированы в естественных изображениях. Поэтому первый слой в нашей модели – это обычный слой `layer_conv_2d()`. Позже мы начнем использовать `layer_separable_conv_2d()`

Мы применяем последовательность сверточных блоков с увеличением глубины признаков. Каждый блок состоит из двух пакетно-нормированных слоев свертки с разделением по глубине и слоя объединения по максимальному соседнему значению с остаточной связью в обход всего блока

```

layer_separable_conv_2d(size, 3, padding = "same", use_bias = FALSE) %>%
layer_batch_normalization() %>%
layer_activation("relu") %>%

layer_separable_conv_2d(size, 3, padding = "same", use_bias = FALSE) %>%
layer_max_pooling_2d(pool_size = 3, strides = 2, padding = "same")

residual <- residual %>%
  layer_conv_2d(size, 1, strides = 2, padding = "same", use_bias = FALSE)

x <- layer_add(list(x, residual))
}

outputs <- x %>%
  layer_global_average_pooling_2d() %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")

model <- keras_model(inputs, outputs)

train_dataset <- image_dataset_from_directory(
  "cats_vs_dogs_small/train",
  image_size = c(180, 180),
  batch_size = 32
)

validation_dataset <- image_dataset_from_directory(
  "cats_vs_dogs_small/validation",
  image_size = c(180, 180),
  batch_size = 32
)

model %>%
  compile(
    loss = "binary_crossentropy",
    optimizer = "rmsprop",
    metrics = "accuracy"
  )

history <- model %>%
  fit(
    train_dataset,
    epochs = 100,
    validation_data = validation_dataset)

```

В исходной модели мы использовали `layer_flatten()` перед `layer_dense()`. Здесь мы используем `layer_global_average_pooling_2d()`

Как и в исходной модели, мы добавляем слой прореживания для регуляризации

Эта сверточная сеть имеет 721 857 параметров, что немного меньше, чем 991 041 параметр исходной модели, которую мы определили в главе 8 (листинг 8.7), но все еще остается примерно на том же уровне. На рис. 9.11 показаны кривые на обучающих и проверочных данных.

Наша новая модель обеспечивает точность на проверочных данных 90,8 % по сравнению с 81,4 % базовой модели из предыдущей главы. Как видите, соблюдение рекомендаций по выбору

архитектуры оказывает немедленное и значительное влияние на точность модели!

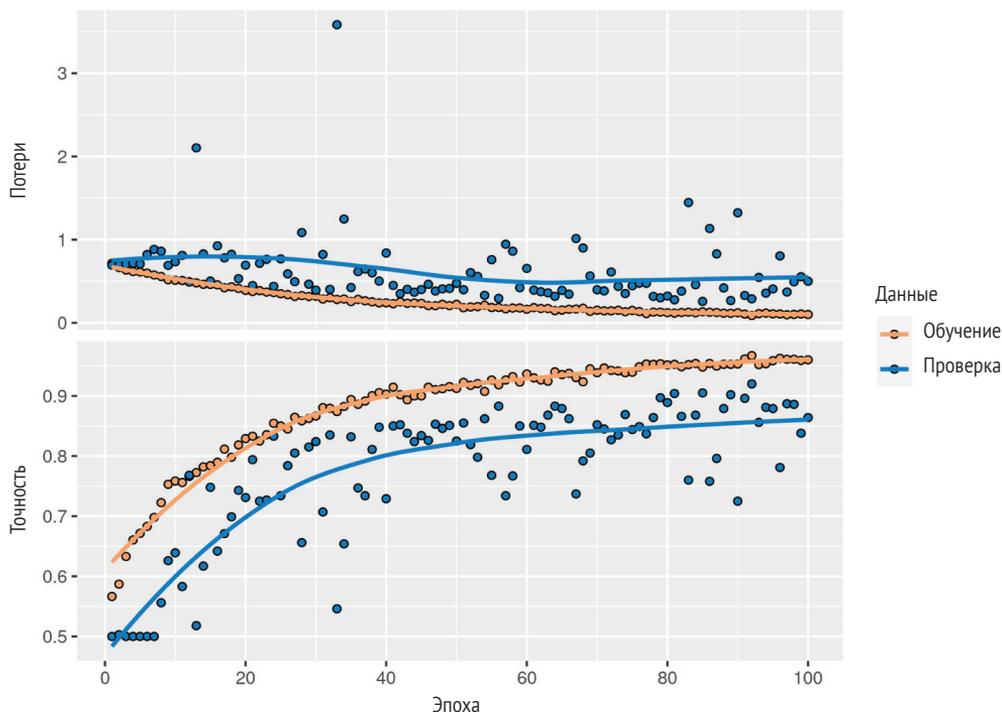


Рис. 9.11 Метрики обучения и проверки модели с архитектурой, подобной Xception

Если вы хотите еще больше повысить точность, на этом этапе вы должны начать систематически настраивать гиперпараметры вашей архитектуры – тема, которую мы подробно рассмотрим в главе 13. Сейчас мы не рассматриваем этот шаг, поэтому конфигурация модели основана исключительно на лучших методиках, которые мы обсуждали, плюс, когда дело доходит до измерения размера модели, немножко интуиции.

Обратите внимание, что рекомендации по выбору архитектуры относятся к компьютерному зрению в целом, а не только к классификации изображений. Например, Xception используется в качестве стандартной сверточной базы в DeepLabV3, популярном современном решении для сегментации изображений<sup>1</sup>.

На этом мы завершаем знакомство с современными базовыми архитектурами сверточных сетей. Используя эти принципы, вы сможете разрабатывать более производительные модели для широкого круга задач компьютерного зрения. Вы уже сделали первые шаги на

<sup>1</sup> Liang-Chieh Chen et al., *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*, ECCV (2018), <https://arxiv.org/abs/1802.02611>.

пути к тому, чтобы стать опытным специалистом по компьютерному зрению. Чтобы углубить ваш опыт, нам нужно раскрыть еще одну важную тему – интерпретацию того, как модель приходит к своим прогнозам.

## 9.4 Интерпретация знаний сверточной нейросети

Фундаментальной проблемой при разработке приложений для компьютерного зрения является *интерпретируемость*: сложно понять, почему ваш классификатор решил, что изображение содержит холодильник, когда вы ясно видите, что это грузовик. Это особенно актуально в сценариях, когда глубокое обучение используется для дополнения человеческого опыта, например для анализа медицинских изображений. В заключительной части этой главы вы познакомитесь с рядом различных методов *визуализации знаний* сверточных сетей и *понимания решений*, которые они принимают.

Часто говорят, что модели глубокого обучения – это «черные ящики»: они изучают представления, которые трудно извлечь и представить в удобочитаемой форме. Хотя это отчасти верно для некоторых типов моделей глубокого обучения, сверточные сети к ним не относятся. Представления, полученные с помощью сверточных сетей, легко поддаются визуализации, в значительной степени потому, что они изначально отражают визуальные понятия. С 2013 года были разработаны разнообразные методы визуализации и интерпретации этих представлений. Мы не будем рассматривать их все, а остановимся на трех самых доступных и полезных:

- *визуализация промежуточных выходных данных сверточной сети (промежуточных активаций)* полезна для понимания того, как последовательные слои преобразуют свои входные данные, и для получения начального представления о значении отдельных фильтров сверточной сети;
- *визуализация фильтров сверточной сети* полезна для точного понимания того, какой визуальный шаблон или понятие воспринимает каждый фильтр в сверточной сети;
- *визуализация тепловых карт активации классов на изображении* полезна для понимания того, какие части изображения были идентифицированы как принадлежащие к определенному классу, что позволяет локализовать объекты на изображениях.

Для иллюстрации первого метода – визуализации промежуточных активаций – мы будем использовать небольшую сеть, которую мы обучили с нуля на задаче классификации собак и кошек в разделе 8.2. Для следующих двух методов мы будем использовать предварительно обученную модель Xception.

## 9.4.1 Визуализация промежуточных активаций

Визуализация промежуточных активаций представляет собой отображение значений, возвращаемых различными слоями свертки и объединения, при заданных входных данных (выход слоя часто называют его *активацией*, или *выходом функции активации*). Это дает представление о том, как ввод разлагается на различные фильтры, изученные сетью. Мы будем визуализировать карты признаков с тремя измерениями: ширина, высота и глубина (каналы). Каждый канал кодирует относительно независимые признаки, поэтому правильный способ визуализации карты признаков – независимо отображать содержимое каждого канала в виде 2D-изображения. Давайте начнем с загрузки модели, которую вы сохранили в разделе 8.2:

```
model <- load_model_tf("convnet_from_scratch_with_augmentation.keras")
model
```

```
Model: "model_3"
```

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 180, 180, 3)]	0
sequential (Sequential)	(None, 180, 180, 3)	0
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d_15 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_9 (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_14 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_8 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_13 (Conv2D)	(None, 41, 41, 128)	73856
max_pooling2d_7 (MaxPooling2D)	(None, 20, 20, 128)	0
conv2d_12 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_6 (MaxPooling2D)	(None, 9, 9, 256)	0
conv2d_11 (Conv2D)	(None, 7, 7, 256)	590080
flatten_3 (Flatten)	(None, 12544)	0
dropout (Dropout)	(None, 12544)	0
dense_3 (Dense)	(None, 1)	12545

=====  
 Total params: 991,041  
 Trainable params: 991,041  
 Non-trainable params: 0

Далее мы готовим входное изображение – снимок кота, который не входит в обучающий набор изображений.

### Листинг 9.6 Предварительная обработка одиночного изображения

```
img_path <- get_file( ← Скачаем тестовое изображение
  fname = "cat.jpg",
  origin = "https://img-datasets.s3.amazonaws.com/cat.jpg")
```

```
img_tensor <- img_path %>%
  tf_read_image(resize = c(180, 180))
```

Прочитаем и изменим размер изображения на тензор формы float32 (180, 180, 3)

Теперь отобразим изображение (рис. 9.12).

### Листинг 9.7 Вывод тестового изображения

```
display_image_tensor (img_tensor)
```



Рис. 9.12 Тестовое изображение кота

Чтобы извлечь карты признаков, подлежащие визуализации, мы создадим модель Keras, которая принимает пакеты изображений в качестве входных данных и выводит активации всех слоев свертки и объединения.

### Листинг 9.8 Создание экземпляра модели, возвращающей активацию слоя

Создаем фиктивные слои conv и pooling, чтобы определить имя класса S3. Обычно это длинная строка, например «keras.layers.convolutional.Conv2D», но поскольку она может меняться между версиями Tensorflow, лучше не задавать ее в коде жестко

```
conv_layer_s3_classname <-
  class(layer_conv_2d(NULL, 1, 1))[1]
pooling_layer_s3_classname <-
  class(layer_max_pooling_2d(NULL))[1]
```

```
is_conv_layer <- function(x) inherits(x, conv_layer_s3_classname)
is_pooling_layer <- function(x) inherits(x, pooling_layer_s3_classname)
```

```
layer_outputs <- list()
for (layer in model$layers)
  if (is_conv_layer(layer) || is_pooling_layer(layer))
    layer_outputs[[layer$name]] <- layer$output
```

Извлекаем выходные данные всех слоев Conv2D и MaxPooling2D и помещаем их в именованный список

```
activation_model <- keras_model(inputs = model$input,
                                outputs = layer_outputs)
```

Создаем модель, которая будет возвращать эти выходные данные для текущего ввода модели

При подаче входного изображения эта модель возвращает значения активаций слоя в исходной модели в виде списка. Вы впервые столкнулись с моделью с несколькими выходами в этой книге на практике с тех пор, как узнали о них в главе 7; до сих пор модели, которые вы видели, имели ровно один вход и один выход. У этой модели есть один вход и девять выходов: по одному выходу на каждую активацию слоя.

### Листинг 9.9 Использование модели для вычисления активаций слоя

Predict() возвращает список из девяти массивов R: один массив на активацию слоя

```
activations <- activation_model %>%
  predict(img_tensor[tf$newaxis, , , ])
```

Вызов `[tf$newaxis, , , ]`, чтобы изменить форму `img_tensor` с `(180, 180, 3)` на `(1, 180, 180, 3)`. Другими словами, добавляем пакетное измерение, поскольку модель ожидает, что входными данными будет пакет изображений, а не одно изображение

Поскольку мы передали именованный список для выходных данных при построении модели, то при вызове для модели функции `predict()` возвращается именованный список массивов R:

```
str(activations)
```

```
List of 9
 $ conv2d_15      : num [1, 1:178, 1:178, 1:32] 0.00418 0.0016 0.00453 0 ...
 $ max_pooling2d_9: num [1, 1:89, 1:89, 1:32] 0.01217 0.00453 0.00742 0.00514
 $ conv2d_14      : num [1, 1:87, 1:87, 1:64] 0 0 0 0 0.00531 ...
 $ max_pooling2d_8: num [1, 1:43, 1:43, 1:64] 0 0 0.00531 0 0 ...
 $ conv2d_13      : num [1, 1:41, 1:41, 1:128] 0 0 0.0288 0 0.0342 ...
 $ max_pooling2d_7: num [1, 1:20, 1:20, 1:128] 0.0313 0.0288 0.0342 0.4004 0.
 $ conv2d_12      : num [1, 1:18, 1:18, 1:256] 0 0 0 0 0 0 0 0 0 ...
 $ max_pooling2d_6: num [1, 1:9, 1:9, 1:256] 0 0 0 0 0 0 0 0 0 ...
 [list output truncated]
```

Давайте подробнее рассмотрим активации первого слоя:

```
first_layer_activation <- activations[[ names(layer_outputs)[1] ]]
dim(first_layer_activation)
```

```
[1] 1 178 178 32
```

Это карта признаков размером 178×178 с 32 каналами. Попробуем визуализировать пятый канал активации первого слоя исходной модели (рис. 9.13).

## Листинг 9.10 Визуализация пятого канала

```

plot_activations <- function(x, ...) {
  x <- as.array(x) ← Преобразование тензоров в массивы

  if(sum(x) == 0)
    return(plot(as.raster("gray")))

  rotate <- function(x) t(apply(x, 2, rev))
  image(rotate(x), asp = 1, axes = FALSE, useRaster = TRUE,
        col = terrain.colors(256), ...)
}

plot_activations(first_layer_activation[, , , 5])

```

Поворот изображения по часовой стрелке  
для удобства просмотра

Каналы со всеми нулевыми значениями  
(т. е. без активаций) отображаются в виде серого  
прямоугольника, поэтому их легко различить



Рис. 9.13 Пятый канал активации первого слоя на тестовом изображении кота

Этот канал, по-видимому, кодирует детектор диагональных краев, но нужно заметить, что ваши собственные каналы могут выглядеть иначе, потому что конкретные фильтры, изученные сверточными слоями, не являются детерминированными.

Теперь построим полную визуализацию всех активаций в сети (рис. 9.14). Мы извлечем и отобразим каждый канал в каждой активации слоя и сложим результаты в одну большую сетку изображений с каналами, расположенными рядом.

## Листинг 9.11 Визуализация каждого канала при каждой промежуточной активации

```

for (layer_name in names(layer_outputs)) { ← Перебор активаций (и имен соответствующих слоев)
  layer_output <- activations[[layer_name]]
}

```

```

n_features <- dim(layer_output) %>% tail(1)
par(mfrow = n2mfrow(n_features, asp = 1.75),
    mar = rep(.1, 4), oma = c(0, 0, 1.5, 0))
for (j in 1:n_features)
  plot_activations(layer_output[, , , j])
  title(main = layer_name, outer = TRUE)
}

```

Активация слоя имеет форму (1, height, width, n\_features)

Подготовка к отображению всех каналов в этой активации на одном рисунке

Это один канал (или функция)

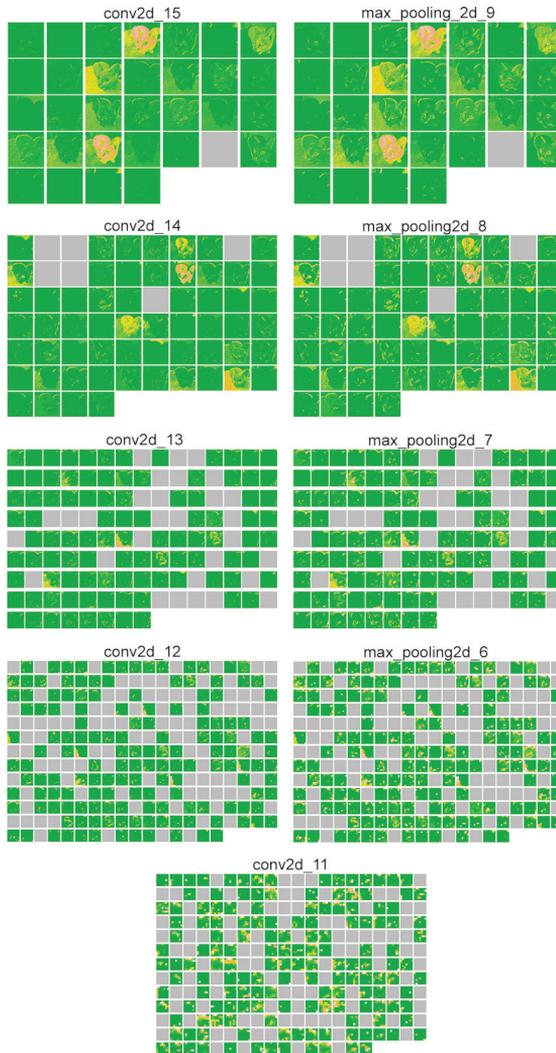


Рис. 9.14 Активация каждого канала каждого слоя на тестовом изображении кота

Здесь следует отметить несколько моментов:

- первый слой действует как набор различных детекторов границ. На этом этапе активации сохраняют почти всю информацию, присутствующую в исходной картинке;

- по мере углубления активации становятся все более абстрактными и менее интерпретируемыми визуально. Они начинают кодировать понятия более высокого уровня, такие как «кошачье ухо» и «кошачий глаз». Более глубокие представления несут все меньше информации о визуальном содержании изображения и все больше – относящейся к классу изображения;
- разреженность активаций увеличивается с глубиной слоя: в первом слое почти все фильтры активируются входным изображением, но в следующих слоях все больше и больше фильтров остаются пустыми. Это означает, что шаблон, закодированный фильтром, не найден во входном изображении.

Мы только что продемонстрировали важную универсальную характеристику представлений, изученных глубокими нейронными сетями: *признаки, извлекаемые слоем, становятся все более абстрактными по мере увеличения глубины слоя*. Активации более высоких слоев несут все меньше и меньше информации о конкретном увиденном входе и все больше информации о цели (в данном случае о классе изображения: кошка или собака). Глубокая нейронная сеть эффективно работает как *конвейер дистилляции информации*, в который поступают необработанные данные (в данном случае RGB-изображения) и многократно преобразуются, чтобы отфильтровать нерелевантную информацию (например, конкретный внешний вид изображения), выделяя и уточняя полезную информацию (например, класс изображения).

Люди и животные воспринимают мир аналогичным образом: понаблюдав за сценой в течение нескольких секунд, человек легко вспомнит, какие ключевые абстрактные объекты в ней присутствовали (велосипед, дерево), но не сможет вспомнить конкретный вид этих объектов. На самом деле, если вы попытаетесь нарисовать обычный велосипед по памяти, скорее всего, вы не сможете сделать это правильно, даже если за свою жизнь вам довелось видеть тысячи велосипедов (например, рис. 9.15). Попробуйте нарисовать велоси-

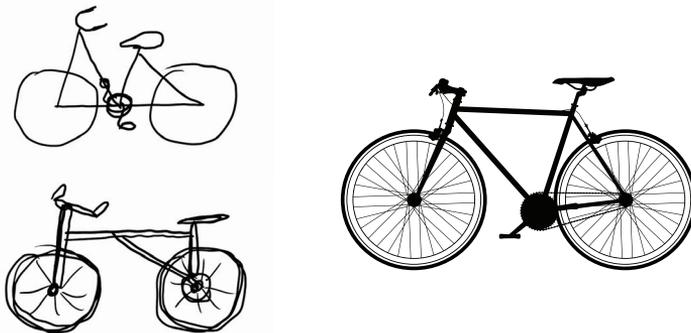


Рис. 9.15 Слева: попытки нарисовать велосипед по памяти; справа: как устроен реальный велосипед

пед прямо сейчас: этот эффект абсолютно реален. Ваш мозг научился полностью абстрагировать свой визуальный ввод – преобразовывая его в высокоуровневые визуальные концепции, отфильтровывая ненужные визуальные детали, – что чрезвычайно затрудняет запоминание того, как выглядят окружающие вас предметы.

## 9.4.2 Визуализация сетевых фильтров

Еще один простой способ проверить фильтры, изученные сверточной сетью, – отобразить характерный визуальный шаблон, на который будет реагировать каждый фильтр. Это можно сделать с помощью *градиентного восхождения во входном пространстве*: применение *градиентного спуска* к входному изображению сверточной сети, чтобы *максимизировать* отклик определенного фильтра, начиная с пустого входного изображения. Результирующее входное изображение будет таким, на которое выбранный фильтр максимально реагирует.

Давайте испытаем этот прием в действии с фильтрами модели Xception, предварительно обученными на ImageNet. Процесс прост: мы построим функцию потерь, которая максимизирует значение фильтра в заданном сверточном слое, а затем будем использовать стохастический градиентный спуск, чтобы настроить значения входного изображения, обеспечивающие наибольшее значение активации. Это будет наш второй пример низкоуровневого цикла градиентного спуска, использующего объект GradientTape() (первый был в главе 2). Сначала создадим модель Xception и загрузим в нее готовые веса, предварительно обученные на наборе данных ImageNet (листинг 9.12).

### Листинг 9.12 Создание сверточной базы Xception

```
model <- application_xception(
  weights = "imagenet",
  include_top = FALSE ←
)
```

Слои классификации не имеют значения для этого варианта использования, поэтому мы не включаем верхний этап модели

Нас интересуют сверточные слои модели – слои Conv2D и SeparableConv2D. Чтобы получить их выводы, нам нужно знать имена соответствующих слоев. Напечатаем их в порядке возрастания глубины (листинг 9.13).

### Листинг 9.13 Печать имен всех сверточных слоев в Xception

```
for (layer in model$layers)
  if(any(grepl("Conv2D", class(layer))))
    print(layer$name)

[1] "block1_conv1"
[1] "block1_conv2"
[1] "block2_sepconv1"
```

```
[1] "block2_sepconv2"
[1] "conv2d_29"

...

[1] "block14_sepconv1"
[1] "block14_sepconv2"
```

Как видите, все слои отдельной свертки `SeparableConv2D` здесь имеют имена наподобие `block6_sepconv1`, `block7_sepconv2` и т. д. `Xception` состоит из блоков, каждый из которых содержит несколько сверточных слоев.

Теперь давайте создадим вторую модель, которая возвращает выходные данные определенного слоя – *экстрактор признаков*. Поскольку наша модель относится к `Functional API`, ее можно просматривать: мы можем запросить выходные данные одного из ее слоев и повторно использовать их в новой модели. Нет необходимости копировать весь код `Xception`.

#### Листинг 9.14 Создание модели экстрактора признаков

Вы можете заменить это именем любого слоя в сверточной базе `Xception`

Это объект слоя, который нас интересует

```
layer_name <- "block3_sepconv1"
layer <- model %>% get_layer(name = layer_name)
feature_extractor <- keras_model(inputs = model$input,
                                outputs = layer$output)
```

Мы используем `model$input` и `layer$output` для создания модели, которая, получая входное изображение, возвращает выходные данные нашего целевого слоя

Для запуска модели достаточно вызвать ее для некоторых входных данных (обратите внимание, что `Xception` требует предварительной обработки входных данных с помощью функции `xception_preprocess_input()`).

#### Листинг 9.15 Использование экстрактора признаков

```
activation <- img_tensor %>%
  .[tf$newaxis, , , ] %>%
  xception_preprocess_input() %>%
  feature_extractor() ←
```

Обратите внимание, что на этот раз мы вызываем модель напрямую, а не с помощью функции `predict()`, и эта активация представляет собой `tf.Tensor`, а не массив `R` (подробнее об этом скоро)

```
str(activation)
```

```
⟨tf.Tensor: shape=(1, 44, 44, 256), dtype=float32, numpy=...⟩
```

Давайте используем нашу модель экстрактора признаков, чтобы найти функцию, которая возвращает скалярное значение, определяющее, насколько данное входное изображение «активирует» данный фильтр в слое. Это своего рода «функция потерь», которую мы *максимизируем* в процессе градиентного подъема:

Функция потерь принимает тензор изображения и индекс рассматриваемого нами фильтра (целое число)

Здесь мы приводим `filter_index` к целочисленному тензору, чтобы убедиться, что у нас есть согласованное поведение, когда мы запускаем эту функцию напрямую (т. е. не через `tf_function()`)

```
compute_loss <- function(image, filter_index) {
  activation <- feature_extractor(image)

  filter_index <- as_tensor(filter_index, "int32")
  filter_activation <-
    activation[, , , filter_index, style = "python"]
  mean(filter_activation[, 3:-3, 3:-3])
}
```

Возвращает среднее значение значений активации для фильтра

Обратите внимание, что мы избегаем краевых артефактов, вовлекая в потери только неграничные пиксели; отбрасываем первые два пикселя по сторонам активации

Необходимо указать, что `filter_index` использует отсчет от нуля, передав параметр `style="python"`

**ПРИМЕЧАНИЕ** Мы будем отслеживать `compute_loss()` с `tf_function()` позже, с `filter_index` в качестве тензоров трассировки. Индексирование в стиле Python (с отсчетом от 0) в настоящее время остается единственным поддерживаемым стилем, причем индекс сам является тензором (это может измениться в будущем). Мы в явном виде указываем, что `filter_index` отсчитывается от нуля при помощи параметра `style = "python"`.

### Разница между `predict(model, x)` и `model(x)`

В предыдущей главе мы использовали для извлечения признаков метод `predict(x)`. Здесь мы используем `model(x)`. Что это дает?

Оба метода, и `y <- predict(model, x)`, и `y <- model(x)` (где `x` – массив входных данных), означают «запустить модель на данных `x` и получить результат `y`». Но все-таки они разные.

Метод `predict()` перебирает данные пакетами (фактически вы можете указать размер пакета в виде `predict(x, batch_size = 64)`) и извлекает значение массива `R` выходных данных. Это схематически эквивалентно следующему коду:

```
predict <- function(model, x) {
  y <- list()
  for(x_batch in split_into_batches(x)) {
    y_batch <- as.array(model(x_batch))
    y[[length(y)+1]] <- y_batch
  }
  unsplit_batches(y)
}
```

Это означает, что вызовы `predict()` могут масштабироваться до очень больших массивов, тогда как `model(x)` выполняется в памяти и не мас-

штабируется. С другой стороны, метод `predict()` не является дифференцируемым: вы не можете получить его градиент, если вызываете его в области видимости `GradientTape()`.

Вы должны использовать `model(x)`, когда вам нужно получить градиенты вызова модели, и вы должны использовать `predict()`, если вам просто нужно выходное значение. Другими словами, всегда используйте `predict()`, если только вы не находитесь в процессе написания низкоуровневого цикла градиентного спуска (как мы сейчас).

Давайте настроим функцию шага градиентного подъема, используя `GradientTape()`. Неочевидный трюк, помогающий процессу градиентного спуска пройти гладко, состоит в том, чтобы нормализовать тензор градиента, разделив его на норму L2 (квадратный корень из среднего квадратов значений в тензоре). Это гарантирует, что величина обновлений, выполняемых для входного изображения, всегда находится в одном и том же диапазоне.

#### Листинг 9.16 Максимизация потерь с помощью стохастического градиентного подъема

Явно наблюдаем за тензором изображения, потому что это не переменная TensorFlow (в градиентной ленте автоматически отслеживаются только переменные)

```

gradient_ascent_step <-
  function(image, filter_index, learning_rate) {
    with(tf$GradientTape() %as% tape, {
      tape$watch(image)
      loss <- compute_loss(image, filter_index)
    })
    grads <- tape$gradient(loss, image)
    grads <- tf$math$l2_normalize(grads)
    image + (learning_rate * grads)
  }

```

Вычисляем скаляр потерь, указывающий, насколько текущее изображение активирует фильтр

Вычисляем градиенты потерь по отношению к изображению

Применяем «трюк нормализации градиента»

Немного переместите изображение в направлении, которое сильнее активирует наш целевой фильтр. Верните обновленное изображение, чтобы мы могли запустить пошаговую функцию в цикле

Теперь у нас есть все необходимые компоненты. Давайте объединим их в функцию R, которая принимает в качестве входных данных имя слоя и индекс фильтра и возвращает тензор, представляющий шаблон, вызывающий максимальную активацию указанного фильтра. Кроме того, мы будем использовать `tf_function()` для ускорения модели.

#### Листинг 9.17 Функция для создания визуализаций фильтра

```

c(img_width, img_height) %<-% c(200, 200)

generate_filter_pattern <- tf_function(function(filter_index) {

```

Инициализация тензора изображения произвольными значениями (модель Xception ожидает входные значения в диапазоне [0, 1], поэтому здесь мы центрируем диапазон к 0,5)

Количество применяемых шагов градиентного восхождения

```

iterations <- 30
learning_rate <- 10
image <- tf$random$uniform(
  minval = 0.4, maxval = 0.6,
  shape = shape(1, img_width, img_height, 3)
)
for (i in seq(iterations))
  image <- gradient_ascent_step(image, filter_index, learning_rate)
image[1, , ]

```

← Величина одного шага

← Циклично обновляем значения тензора изображения в стремлении максимизировать функцию потерь

← Сокращаем разрядность и выводим изображение

Результирующий тензор изображения представляет собой массив с плавающей запятой формы (200, 200, 3) со значениями, которые могут и не быть целыми числами в пределах [0, 255]. Следовательно, нам нужно выполнить постобработку тензора, чтобы превратить его в отображаемое изображение. Для этого мы возьмем тензорные операции, сформируем вспомогательную функцию и обернем ее в `tf_function()`, чтобы ускорить процесс.

### Листинг 9.18 Вспомогательная функция для преобразования тензора в допустимое изображение

```

deprocess_image <- tf_function(function(image, crop = TRUE) {
  image <- image - mean(image)
  image <- image / tf$math$reduce_std(image)
  image <- (image * 64) + 128
  image <- tf$clip_by_value(image, 0, 255)
  if(crop)
    image <- image[26:-26, 26:-26, ]
  image
})

```

← Нормализация значений изображения к диапазону [0, 255]

← Обрезка по центру, чтобы избежать артефактов на границах

`mean()` вызывает `tf$math$reduce_mean()`

Попробуем функцию в деле (рис. 9.16):

```

generate_filter_pattern(filter_index = as_tensor(2L)) %>%
  deprocess_image() %>%
  display_image_tensor()

```

Обратите внимание, что здесь мы используем `filter_index` с методом `as_tensor()`. Мы делаем это потому, что `tf_function()` компилирует отдельную оптимизированную функцию для каждого уникального способа ее вызова, а константный литерал является уникальной сигнатурой вызова. Если бы мы не вызвали здесь метод `as_tensor()`, то в следующем цикле, где мы отрисовываем первые 64 активации, `tf_function()` отследил бы и скомпилировал

`generate_filter_pattern()` 64 раза! Однако вызов декорированной функции `tf_function()` с тензором, даже константным, не является уникальной сигнатурой функции для `tf_function()`, поэтому `generate_filter_pattern()` трассируется только один раз.

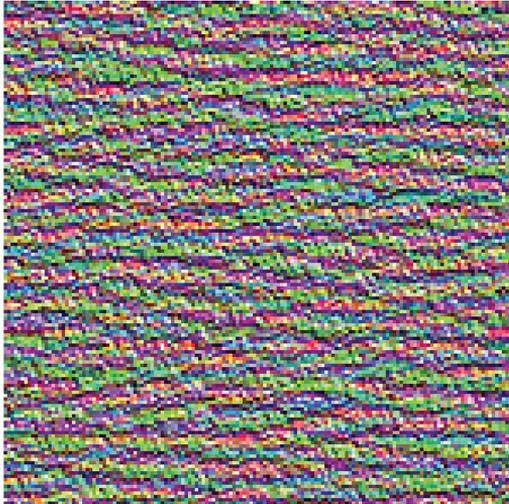


Рис. 9.16 Паттерн, на который максимально реагирует второй канал в слое `block3_sepconv1`

Судя по изображению, третий фильтр в слое `block3_sepconv1` реагирует на рисунок из горизонтальных линий, чем-то похожий на воду или мех.

Теперь самое интересное: вы можете начать визуализировать каждый фильтр в слое и даже каждый фильтр в каждом слое модели.

#### Листинг 9.19 Создание сетки всех шаблонов наибольшего отклика фильтра в слое

```

par(mfrow = c(8, 8))
for (i in seq(0, 63)) {
  generate_filter_pattern(filter_index = as_tensor(i)) %>%
  deprocess_image() %>%
  display_image_tensor(plot_margins = rep(.1, 4))
}

```

Создание и построение визуализаций для первых 64 фильтров в слое

Эти визуализации фильтров (рис. 9.17) многое говорят нам о том, как слои сверточной сети видят мир: каждый слой изучает набор фильтров, так что их входные данные могут быть выражены как комбинация фильтров. Это похоже на то, как преобразование Фурье разлагает сигналы на частотные составляющие. Фильтры в этих наборах фильтров становятся все более сложными и совершенными по мере продвижения вглубь модели:

- фильтры из первых слоев модели кодируют простые направленные края и цвета (или в некоторых случаях цветные края);

- фильтры из слоев чуть выше по стеку, например `block4_sepconv1`, кодируют простые текстуры, состоящие из комбинаций краев и цветов;
- фильтры в более высоких слоях начинают напоминать текстуры, встречающиеся в естественных изображениях: перья, глаза, листья и т. д.

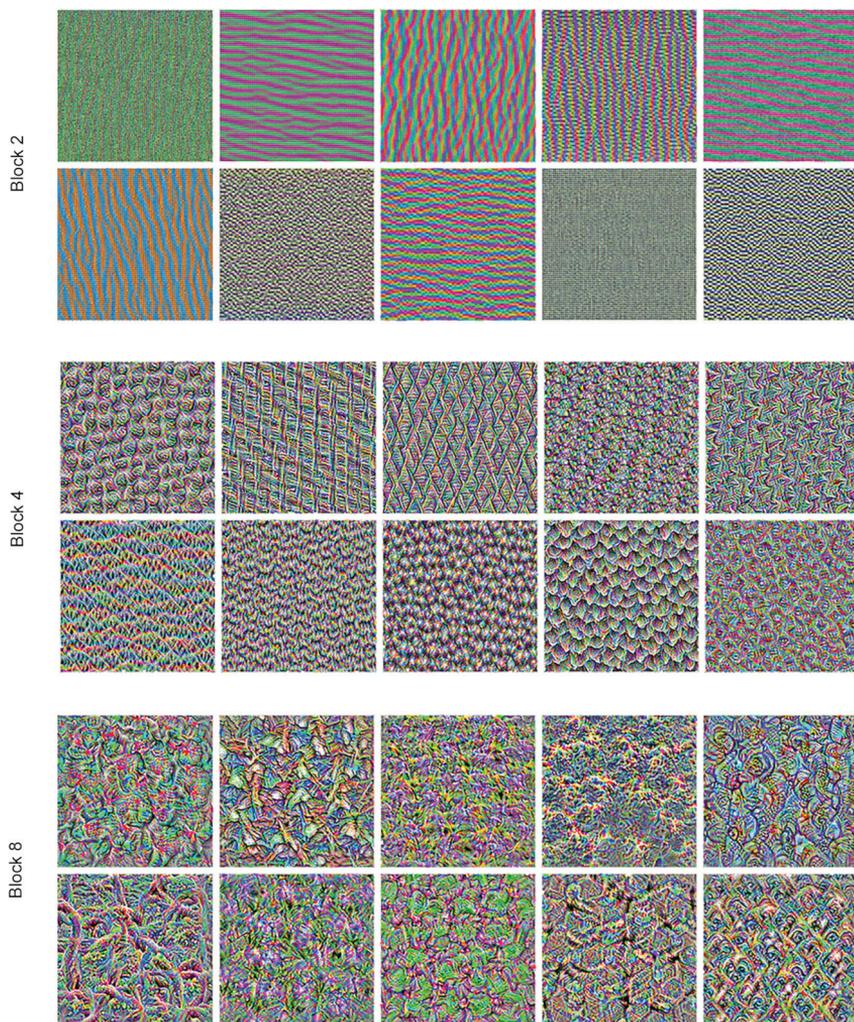


Рис. 9.17 Некоторые шаблоны фильтров для слоев `block2_sepconv1`, `block4_sepconv1` и `block8_sepconv1`

### 9.4.3 Визуализация тепловых карт активации класса

В этом разделе вы познакомитесь с последним методом визуализации, который полезен для понимания того, какие части данного

изображения привели сверточную сеть к окончательному классификационному решению. Это полезно для «отладки» процесса принятия решения в сети, особенно в случае ошибочной классификации (область исследований, называемая *интерпретируемостью модели*). Данный метод также позволяет найти определенные объекты на изображении.

Эта общая категория методов называется визуализацией *карты активации класса* (class activation map, CAM) и заключается в создании тепловых карт активации класса по входным изображениям. *Тепловая карта активации класса* – это двумерная сетка оценок, связанных с конкретным выходным классом, рассчитанная для каждого местоположения заданного изображения и указывающая, насколько важно каждое местоположение по отношению к рассматриваемому классу. Например, при наличии изображения, загруженного в сеть классификации «собака или кошка», CAM-визуализация позволит вам сгенерировать тепловую карту для класса «кошка», показывающую, насколько «кошачьими» являются различные части изображения, а также тепловую карту для класса «собака», указывающую на части изображения, характерные для собаки.

Конкретная реализация, которую мы будем использовать, описана в статье под названием «Grad-CAM: визуальное толкование работы глубоких сетей с помощью локализации на основе градиента»<sup>1</sup>.

Рабочий процесс Grad-CAM состоит из двух основных этапов: получения выходной карты признаков сверточного слоя для имеющегося входного изображения и взвешивания каждого канала в этой карте признаков с помощью градиента класса по отношению к каналу. Один из способов понять этот подход – представить, что вы взвешиваете пространственную карту того, «насколько интенсивно входное изображение активирует различные каналы», по критерию «насколько важен каждый канал по отношению к классу», в результате чего получается пространственная карта «насколько сильно входное изображение активирует класс». Давайте продемонстрируем эту технику, используя предварительно обученную модель Xception.

#### Листинг 9.20 Загрузка сети Xception с предварительно обученными весами

```
model <- application_xception(weights = "imagenet")
```

Обратите внимание, что здесь включен полносвязный классификатор сверху; во всех предыдущих случаях мы его отбрасывали

В качестве примера возьмем изображение двух африканских слонов, показанных на рис. 9.18, возможно, матери и ее детеныша, гуляющих по саванне. Преобразуем это изображение в объект, который может прочитать модель Xception: модель была обучена на изображениях размером 299×299, предварительно обработанных в соот-

<sup>1</sup> Ramprasaath R. Selvaraju et al., arXiv (2017), <https://arxiv.org/abs/1610.02391>.

ветствии с несколькими правилами, которые упакованы в служебную функцию `xception_preprocess_input()`.



Рис. 9.18 Тестовое изображение африканских слонов

Итак, нам нужно загрузить исходное изображение, изменить его размер до  $299 \times 299$ , преобразовать в тензор `float32` и применить правила предварительной обработки.

#### Листинг 9.21 Предварительная обработка входного изображения для Xception

Загрузите образ и сохраните его локально по пути `img_path`

```
img_path <- get_file(
  fname = "elephant.jpg",
  origin = "https://img-datasets.s3.amazonaws.com/elephant.jpg")
img_tensor <- tf_read_image(img_path, resize = c(299, 299))
preprocessed_img <- img_tensor[tf$newaxis, , ] %>%
```

Предварительно обработайте пакет (это нормализация цвета по каналам)

Добавьте измерение, чтобы преобразовать массив в партию размера (1, 299, 299, 3)

```
xception_preprocess_input()
```

Прочитайте изображение как тензор и измените его размер до  $299 \times 299$ .

`img_tensor` – это `float32` с формой (299, 299, 3)

Теперь вы можете запустить предварительно обученную сеть на изображении и декодировать ее прогнозный вектор обратно в удобочитаемый формат:

```
preds <- predict(model, preprocessed_img)
str(preds)
```

```
num [1, 1:1000] 0.00000551 0.00002746 0.00001734 0.00001188 0.00001152 ...
```

```
imagenet_decode_predictions(preds, top=3)[[1]]
```

```
class_name class_description      score
1 n02504458 African_elephant 0.90519804
2 n01871265                tusker 0.05259838
3 n02504013  Indian_elephant 0.01615972
```

Три наиболее вероятных класса, предсказанных для этого изображения, следующие:

- африканский слон (с вероятностью 90 %);
- рабочий слон с большими бивнями (с вероятностью 5 %);
- индийский слон (с вероятностью 2 %).

Таким образом, сеть пришла к выводу, что изображение содержит неопределенное количество африканских слонов. Запись в векторе предсказания, которая была максимально активирована, соответствует классу «африканский слон» с индексом 387:

```
which.max(preds[1, ])
```

```
[1] 387
```

Чтобы выяснить, какие части изображения, по мнению сети, больше всего похожи на африканских слонов, давайте построим процесс Grad-CAM. Сначала создадим модель, которая сопоставляет входное изображение с активациями последнего сверточного слоя (листинг 9.22).

#### Листинг 9.22 Настройка модели, возвращающей последний результат свертки

```
last_conv_layer_name <- "block14_sepconv2_act"
classifier_layer_names <- c("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)
```

Имена последних двух слоев в модели Xception

Вторым шагом создадим модель, которая сопоставляет активацию последнего сверточного слоя с окончательными предсказаниями класса (листинг 9.23).

#### Листинг 9.23 Повторное применение классификатора поверх последнего вывода свертки

```
classifier_input <- layer_input(batch_shape = last_conv_layer$output$shape)
x <- classifier_input
```

```
for (layer_name in classifier_layer_names)
  x <- get_layer(model, layer_name)(x)

classifier_model <- keras_model(classifier_input, x)
```

Теперь вычислим градиент верхнего предсказанного класса для нашего входного изображения по отношению к активациям последнего слоя свертки.

**Листинг 9.24 Получение градиентов верхнего предсказанного класса**

Вычисляем активации последнего конверсионного слоя и заставляем ленту следить за ним

Получаем канал активации, соответствующий верхнему предсказанному классу

```
with (tf$GradientTape() %as% tape, {
  last_conv_layer_output <- last_conv_layer_model(preprocessed_img)
  tape$watch(last_conv_layer_output)
  preds <- classifier_model(last_conv_layer_output)
  top_pred_index <- tf$argmax(preds[1, ])
  top_class_channel <- preds[, top_pred_index, style = "python"]
})

grads <- tape$gradient(top_class_channel, last_conv_layer_output)
```

Этот градиент верхнего предсказанного класса по отношению к выходной карте объектов последнего сверточного слоя

Применим объединение и взвешивание по важности к тензору градиента, чтобы получить тепловую карту активации класса.

**Листинг 9.25 Объединение градиентов и взвешивание важности каналов**

Здесь мы используем правила преобразования размерности тензоров, чтобы избежать написания цикла for. Оси размера 1 преобразуются, чтобы соответствовать соответствующим осям last\_conv\_layer\_output

pooled\_grads – это вектор, каждая запись которого представляет собой среднюю интенсивность градиента для данного канала. Он количественно определяет важность каждого канала по отношению к высшему прогнозируемому классу

```
pooled_grads <- mean(grads, axis = c(1, 2, 3), keepdims = TRUE)

heatmap <- (last_conv_layer_output * pooled_grads) %>%
  mean(axis = -1) %>%
  .[, , ]
```

Среднее значение полученной карты признаков по каналам – это наша тепловая карта активации класса

Умножьте каждый канал на выходе последнего сверточного слоя на «насколько важен этот канал»

Градиент и last\_conv\_layer\_output имеют одинаковую форму (1, 10, 10, 2048)

Форма: (1, 10, 10, 2048)

Форма: (1, 10, 10)

Отбрасываем измерение пакетов; форма выхода: (10, 10)

pooled\_grads имеет форму (1, 1, 1, 2048)

Результат показан на рис. 9.19.

## Листинг 9.26 Постобработка тепловой карты

```
par(mar = c(0, 0, 0, 0))
plot_activations(heatmap)
```

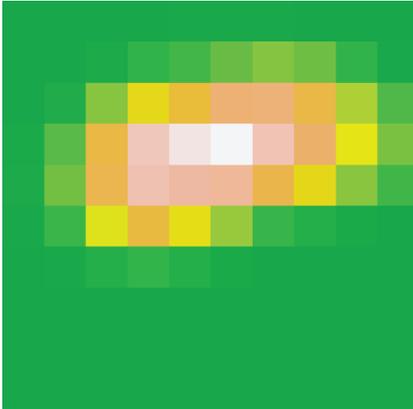


Рис. 9.19 Тепловая карта активации автономного класса

Наконец, наложим тепловую карту активаций на исходное изображение. Мы вырезаем методом `cut()` значения тепловой карты `heatmap` в последовательную цветовую палитру, а затем конвертируем в растровый объект `raster` языка R. Обязательно нужно передать палитре параметр `alpha = .4`, чтобы мы могли видеть исходное изображение, когда накладываем на него тепловую карту (рис. 9.20.)

## Листинг 9.27 Наложение тепловой карты на исходное изображение

```
pal <- hcl.colors(256, palette = "Spectral", alpha = .4, rev = TRUE)
heatmap <- as.array(heatmap)
heatmap[,] <- pal[cut(heatmap, 256)]
heatmap <- as.raster(heatmap)
```

Загружаем исходное изображение, на этот раз без масштабирования

```
img <- tf_read_image(img_path, resize = NULL)
display_image_tensor(img)
rasterImage(heatmap, 0, 0, ncol(img), nrow(img), interpolate = FALSE)
```

Накладываем тепловую карту на исходное изображение с непрозрачностью 40%. Мы передаем `ncol(img)` и `nrow(img)`, чтобы тепловая карта с меньшим количеством пикселей отображалась в соответствии с размером исходного изображения. Мы передаем `interpolate = FALSE`, чтобы мы могли ясно видеть, где находятся границы пикселей карты активации

Эта техника визуализации отвечает на два важных вопроса:

- почему сеть решила, что на этом изображении африканский слон?
- где на картинке находится африканский слон?

В частности, интересно отметить, что область, соответствующая ушам слоненка, сильно активирована: вероятно, по этому признаку сеть может отличить африканских слонов от индийских.

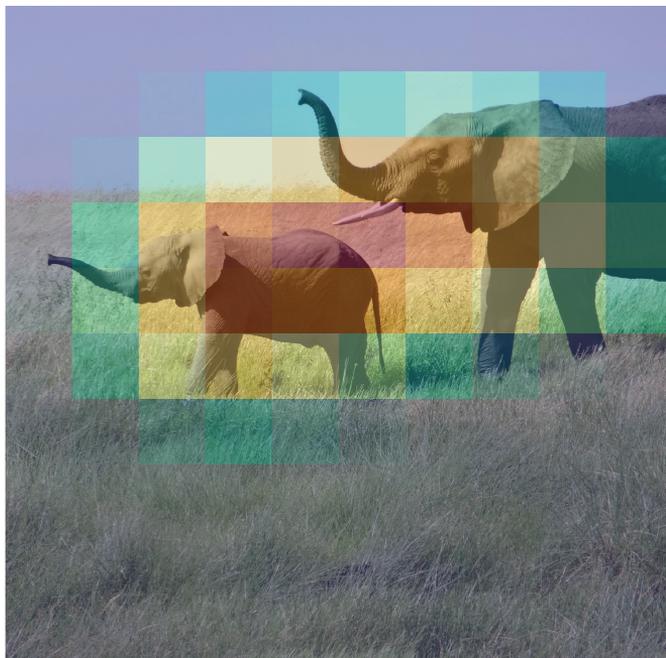


Рис. 9.20 Тепловая карта активации класса африканских слонов на тестовом изображении

## Краткие итоги главы

- С помощью глубокого обучения вы можете реализовать три основные функции компьютерного зрения: классификацию изображений, сегментацию изображений и обнаружение объектов.
- Выполнение рекомендаций по разработке современной архитектуры сверточных сетей поможет вам получить максимальную отдачу от ваших моделей. Некоторые из лучших методик включают использование остаточных связей, пакетную нормализацию и свертки с разделением по глубине.
- Представления, которые изучают свертки, легко проверить – сверточные сети не являются черными ящиками!
- Вы можете создавать визуализации фильтров, изученных вашими сверточными сетями, а также тепловые карты активности класса.

# 10

## Глубокое обучение и временные ряды

---

### **Эта глава охватывает следующие темы:**

- примеры задач машинного обучения, использующих данные временных рядов;
- принцип работы рекуррентных нейронных сетей (RNN);
- применение RNN на примере прогнозирования температуры;
- улучшенные методы использования RNN.

### **10.1 Различные виды задач временных рядов**

*Временной ряд* может представлять собой любые данные, полученные путем измерений через равные промежутки времени, например дневные колебания цены акций, почасовое потребление электроэнергии в городе или еженедельные продажи в магазине. Временные ряды есть везде, независимо от того, рассматриваем ли мы природные явления (например, сейсмическую активность, изменения популяций рыб в реке или погоду в каком-либо месте) или модели человеческой деятельности (допустим, количество посетителей веб-сайта, ВВП страны или объем операций с кредитными картами). В отличие от типов данных, с которыми вы сталкивались до сих пор,

работа с временными рядами включает в себя понимание *динамики* системы – ее периодических циклов, тенденций во времени, регулярного режима и внезапных всплесков.

На сегодняшний день наиболее распространенной задачей, связанной с временными рядами, является *прогнозирование во времени* (forecasting) – предсказание потребления электроэнергии на несколько часов вперед, чтобы можно было предвидеть спрос и настроить генерацию; прогнозирование доходов на несколько месяцев вперед, чтобы вы могли планировать свой бюджет; прогноз погоды на несколько дней вперед, чтобы вы могли планировать свои выходные. Прогнозирование во времени – тема этой главы. Но на самом деле есть множество других полезных вещей, которые вы можете делать с временными рядами:

- *классификация* – присвоение временному ряду одной или нескольких категориальных меток. Например, анализируя временной ряд активности посетителя на веб-сайте, можно сделать вывод, является ли посетитель ботом или человеком;
- *обнаружение событий* – выявление определенного ожидаемого события в непрерывном потоке данных. Особенно полезным является обнаружение «горячих слов», когда модель отслеживает аудиопоток и обнаруживает такие высказывания, как «ОК, Google» или «Привет, Алиса»;
- *обнаружение аномалий* – обнаружение любых необычных явлений в непрерывном потоке данных. Необычная активность в вашей корпоративной сети? Возможно, это атака хакеров. Необычные показания датчиков на производственной линии? Нужно пойти и узнать, в чем дело. Обнаружение аномалий обычно осуществляется с помощью обучения без учителя, потому что вы часто не знаете наперед, какую аномалию ищете, следовательно, не можете обучить модель на конкретных примерах аномалий.

При работе с временными рядами вы столкнетесь с широким спектром методов представления данных, специфичных для предметной области. Например, вы наверняка слышали о преобразовании Фурье, которое заключается в представлении сигнала сложной формы в виде суперпозиции колебаний разных частот. Преобразование Фурье имеет очень большое значение, особенно при предварительной обработке любых данных, которые в первую очередь характеризуются своими циклами и колебаниями (например, звук, колебания каркаса небоскреба или ваши мозговые волны). В контексте глубокого обучения анализ Фурье (или связанный с ним *мелкочастотный спектральный анализ*) и другие представления, специфичные для предметной области, могут быть полезны как способ конструирования признаков или подготовки более эффективных обучающих данных. Однако мы не будем рассматривать эти методы в данной книге и сосредоточимся только на моделировании.

В этой главе вы узнаете о *рекуррентных нейронных сетях* (recurrent neural networks, RNN) и применении их для прогнозирования временных рядов.

## 10.2 Пример прогнозирования температуры

На протяжении этой главы все наши примеры кода будут нацелены на одну задачу: прогнозирование температуры на 24 часа вперед, учитывая временной ряд ежечасных измерений таких величин, как атмосферное давление и влажность, полученных при помощи датчиков на крыше здания. Очевидно, что это довольно сложная задача!

Мы воспользуемся этой задачей прогнозирования температуры, чтобы выделить аспекты, принципиально отличающие временные ряды от других разновидностей наборов данных, с которыми вы сталкивались до сих пор.

Вы убедитесь, что полносвязные и сверточные сети плохо приспособлены для работы с наборами данных такого рода, в то время как рекуррентные нейронные сети блестяще справляются с прогнозированием временных рядов.

Мы будем работать с набором временных рядов данных о погоде, записанных на метеостанции Института биогеохимии Макса Планка в Йене, Германия<sup>1</sup>. В этом наборе данных 14 различных величин (таких как температура, давление, влажность и направление ветра) регистрировались каждые 10 минут в течение нескольких лет. Исходные данные восходят к 2003 году, но подмножество данных, которые мы загрузим, ограничено 2009–2016 годами. Начнем с загрузки и распаковки данных:

```
url <-
  "https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip"
download.file(url, destfile = basename(url))
zip::unzip(zipfile = "jena_climate_2009_2016.csv.zip",
           files = "jena_climate_2009_2016.csv")
```

Теперь посмотрим, как выглядят данные. Мы будем использовать `readr::read_csv()` для чтения данных.

### Листинг 10.1 Просмотр набора данных о погоде в Йене

```
full_df <- readr::read_csv("jena_climate_2009_2016.csv")
```

Вы можете пропустить приведенный выше вызов `zip::unzip()` и передать путь к zip-файлу напрямую в `read_csv()`

Эта строка кода выводит на экран фрейм данных с 420 451 строкой и 15 столбцами. Каждая строка представляет собой точку на оси времени: запись даты и 14 значений, связанных с погодой.

<sup>1</sup> Adam Erickson, Olaf Kolle, <http://www.bgc-jena.mpg.de/wetter>.

```
full_df
```

```
# A tibble: 420,451 × 15
  `Date Time`      `p (mbar)` `T (degC)` `Tpot (K)` `Tdew (degC)` `rh (%)`
  <chr>            <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
1 01.01.2009 00:10... 997.     -8.02     265.     -8.9     93.3
2 01.01.2009 00:20... 997.     -8.41     265.     -9.28    93.4
3 01.01.2009 00:30... 997.     -8.51     265.     -9.31    93.9
4 01.01.2009 00:40... 997.     -8.31     265.     -9.07    94.2
5 01.01.2009 00:50... 997.     -8.27     265.     -9.04    94.1
6 01.01.2009 01:00... 996.     -8.05     265.     -8.78    94.4
7 01.01.2009 01:10... 996.     -7.62     266.     -8.3     94.8
8 01.01.2009 01:20... 996.     -7.62     266.     -8.36    94.4
9 01.01.2009 01:30... 996.     -7.91     266.     -8.73    93.8
10 01.01.2009 01:40... 997.     -8.43     265.     -9.34    93.1
# ... with 420,441 more rows, and 9 more variables: `VPmax (mbar)` <dbl>,
# `VPact (mbar)` <dbl>, `VPdef (mbar)` <dbl>, `sh (g/kg)` <dbl>,
# `H2OC (mmol/mol)` <dbl>, `rho (g/m**3)` <dbl>, `wv (m/s)` <dbl>,
# `max. wv (m/s)` <dbl>, `wd (deg)` <dbl>
```

Метод `read_csv()` правильно распознал все столбцы как числовые векторы, за исключением столбца "Date Time", который он проанализировал как символьный вектор, а не как вектор даты и времени. Мы не будем использовать столбец "Date Time" для обучения модели, так что это не проблема, но просто для полноты картины мы можем преобразовать столбец символов в формат R POSIXct. Нужно отметить, что мы передаем `tz = "Etc/GMT+1"` вместо `tz = "Europe/Berlin"`, потому что временные метки в наборе данных не переключаются на средневропейское летнее время, а вместо этого всегда соответствуют центральноевропейскому времени:

```
full_df$`Date Time` %<>%
  as.POSIXct(tz = "Etc/GMT+1", format = "%d.%m.%Y %H:%M:%S")
```

### Оператор конвейера присваивания %<>%

В предыдущей строке кода мы впервые использовали конвейер присваивания. Запись `x %<>% fn()` является сокращением для `x <- x %>% fn()`. Она позволяет писать более читаемый код и избегать многократного повторения одного и того же имени переменной. Вместо конвейера присваивания мы могли бы с тем же результатом написать:

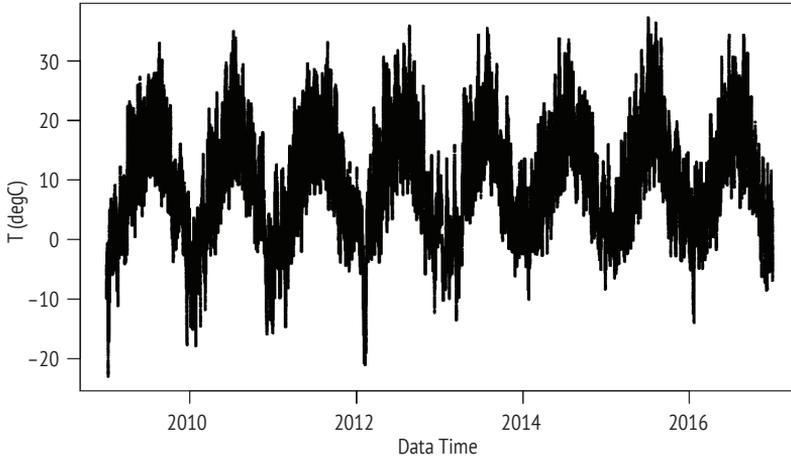
```
full_df$`Date Time` <- full_df$`Date Time` %>%
  as.POSIXct(tz = "Etc/GMT+1", format = "%d.%m.%Y %H:%M:%S")
```

Конвейер присваивания становится доступным при вызове `library(keras)`.

На рис. 10.1 показан график зависимости температуры (в градусах Цельсия) от времени. На этом графике хорошо видна годовая периодичность температуры – данные охватывают 8 лет.

**Листинг 10.2** Построение графика временного ряда температуры

```
plot(`T (degC)` ~ `Date Time`, data = full_df, pch = 20, cex = .3)
```

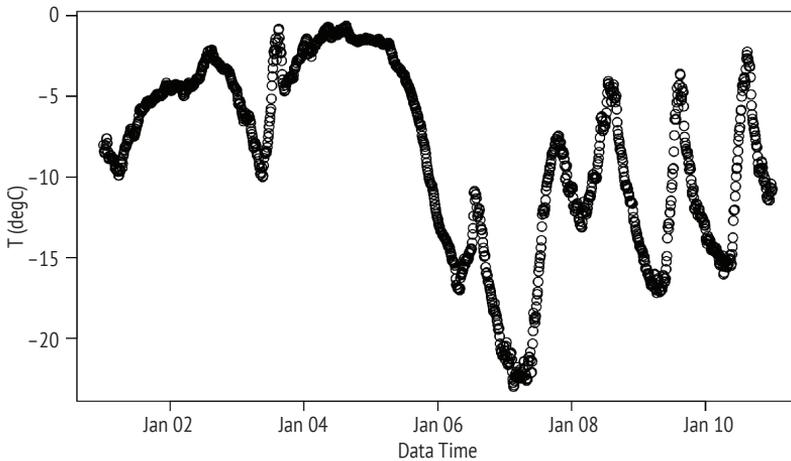


**Рис. 10.1** Температура во всем временном диапазоне набора данных (°C)

На рис. 10.2 показан усеченный график данных температуры за первые 10 дней. Поскольку данные записываются каждые 10 минут, вы получаете  $24 \times 6 = 144$  измерения в день.

**Листинг 10.3** Построение графика первых 10 дней временного ряда температуры

```
plot(`T (degC)` ~ `Date Time`, data = full_df[1:1440, ])
```



**Рис. 10.2** Температура за первые 10 дней набора данных (°C)

На этом графике вы можете видеть суточный цикл, особенно четко выраженный в данных за последние четыре дня. Также стоит отметить, что этот 10-дневный период приходится на довольно холодный зимний месяц.

### Всегда ищите периодичность в ваших данных

Периодичность в нескольких масштабах шкалы времени является важным и очень распространенным свойством данных временных рядов. Независимо от того, анализируете ли вы погоду, занятость парковки в торговом центре, посещаемость веб-сайта, продажи в продуктовом магазине или шаги, зарегистрированные в фитнес-трекере, вы увидите, как минимум, ежедневные и годовые циклы (данные, созданные человеком, также часто содержат недельные циклы). При изучении ваших данных обязательно ищите эти 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. Но изменение температуры в масштабе нескольких дней выглядит более хаотичным. Можно ли с высокой надежностью предсказать временную последовательность в масштабе суток? Давайте посмотрим.

Во всех наших экспериментах мы будем использовать первые 50 % данных для обучения, следующие 25 % для проверки и последние 25 % для контроля. При работе с данными временных рядов принципиально важно использовать контрольные и проверочные данные, которые являются более свежими, чем обучающие, потому что вы пытаетесь предсказать будущее, учитывая прошлое, а не наоборот, и разделение данных на поднаборы должно отражать этот факт. Некоторые задачи окажутся значительно проще, если вы направите ось времени из будущего в прошлое, но нам такое жульничество не подходит!

#### Листинг 10.4 Вычисление количества выборок, которые мы будем использовать для каждого разделения данных

```
num_train_samples <- round(nrow(full_df) * .5)
num_val_samples <- round(nrow(full_df) * 0.25)
num_test_samples <- nrow(full_df) - num_train_samples - num_val_samples

train_df <- full_df[seq(num_train_samples), ] ← Первые 50 % строк, 1:210226

val_df <- full_df[seq(from = nrow(train_df) + 1, ← Следующие
                    length.out = num_val_samples), ] 25 % строк,
                                                       210227:315339

test_df <- full_df[seq(to = nrow(full_df), ← Последние 25 % строк, 315340:420451
                    length.out = num_test_samples), ]
```

```
cat("num_train_samples:", nrow(train_df), "\n")
cat("num_val_samples:", nrow(val_df), "\n")
cat("num_test_samples:", nrow(test_df), "\n")
```

```
num_train_samples: 210226
num_val_samples: 105113
num_test_samples: 105112
```

## 10.2.1 Подготовка данных

Точная постановка задачи будет следующей: предсказать температуру через 24 часа по данным, охватывающим предыдущие пять дней и замеряемым один раз в час.

Первым делом необходимо преобразовать данные в формат, понятный нейронной сети. Это легко: данные уже представлены в числовом виде, поэтому нам не придется как-то векторизовать их. Однако все временные последовательности в данных имеют разный масштаб (например, атмосферное давление, измеряемое в миллибарах, изменяется около значения 1000, тогда как параметр H2O, измеряемый в миллимолях на моль, изменяется вблизи значения 3). Необходимо нормализовать временные последовательности независимо друг от друга, чтобы все они состояли из небольших по величине значений примерно одинакового масштаба. Мы собираемся использовать в качестве обучающих данных первые 210 226 записей, поэтому будем вычислять среднее значение и стандартное отклонение только для этого пакета.

### Листинг 10.5 Нормализация данных

```
input_data_colnames <- names(full_df) %>%
  setdiff(c("Date Time"))
```

← На вход модели подаются все столбцы данных, кроме Date Time

```
normalization_values <-
  zip_lists(mean = lapply(train_df[input_data_colnames], mean),
            sd = lapply(train_df[input_data_colnames], sd))
```

← Вычисляем коэффициенты нормализации, используя только обучающие данные!

```
str(normalization_values)
```

```
List of 14
 $ p (mbar) :List of 2
  ..$ mean: num 989
  ..$ sd : num 8.51
 $ T (degC) :List of 2
  ..$ mean: num 8.83
  ..$ sd : num 8.77
 $ Tpot (K) :List of 2
  ..$ mean: num 283
  ..$ sd : num 8.87
 $ Tdew (degC) :List of 2
  ..$ mean: num 4.31
  ..$ sd : num 7.08
```

```

$ rh (%) :List of 2
..$ mean: num 75.9
..$ sd : num 16.6
$ VPmax (mbar) :List of 2
..$ mean: num 13.1
..$ sd : num 7.6
$ VPact (mbar) :List of 2
..$ mean: num 9.19
..$ sd : num 4.15
$ VPdef (mbar) :List of 2
..$ mean: num 3.95
..$ sd : num 4.77
[list output truncated]

```

```

normalize_input_data <- function(df) {
  normalize <- function(x, center, scale) ←
    (x - center) / scale

  for(col_nm in input_data_colnames) {
    col_nv <- normalization_values[[col_nm]]
    df[[col_nm]] %<>% normalize(., col_nv$mean, col_nv$sd)
  }
  df
}

```

Вместо этого вы также можете вызвать `scale(col, center = train_col_mean, scale = train_col_sd)`, но для максимальной наглядности мы определяем локальную функцию `normalize()`

Теперь создадим объект набора данных TensorFlow, который выдает пакеты данных за последние пять дней вместе с целевой температурой через 24 часа в будущем. Поскольку выборки в наборе данных сильно избыточны (выборка  $N$  и выборка  $N + 1$  будут иметь общие временные шаги), было бы расточительно напрямую выделять память для каждой выборки. Вместо этого мы будем генерировать выборки на ходу, сохраняя в памяти только исходные массивы данных и ничего более.

Не составит труда написать функцию R для этой операции, но в Keras есть встроенная утилита для работы с набором данных `timeseries_dataset_from_array()`, которая генерирует выборки по ходу выполнения, поэтому мы можем сэкономить время на разработку кода. В дальнейшем вы сможете использовать эту утилиту практически для любой задачи прогнозирования временных рядов.

### Подробнее про `timeseries_dataset_from_array()`

Чтобы лучше понять, что делает `timeseries_dataset_from_array()`, давайте рассмотрим простой пример. Основная идея этой утилиты заключается в том, что вы предоставляете массив данных временных рядов (аргумент `data`), а `timeseries_dataset_from_array()` возвращает *окна*, извлеченные из исходных временных рядов (мы будем называть их выборками).

Например, если вы используете `data = [0 1 2 3 4 5 6]` и `sequence_length = 3`, то `timeseries_dataset_from_array()` сгенерирует следующие выборки: `[0 1 2]`, `[1 2 3]`, `[2 3 4]`, `[3 4 5]`, `[4 5 6]`.

Вы также можете передать в `timeseries_dataset_from_array()` аргумент `targets` (массив). Первая запись массива `targets` должна соответствовать желаемой цели для первой последовательности, которая будет сгенерирована из массива данных. Итак, если вы делаете прогнозирование временных рядов, аргумент `targets` должны быть тем же массивом, что и `data`, со смещением на некоторую величину.

Например, с `data = [0 1 2 3 4 5 6 ...]` и `sequence_length = 3` вы можете создать набор данных для прогнозирования следующего шага в серии, передав аргумент `targets = [3 4 5 6 ...]`. Попробуем это сделать:

```

Генерируем массив упорядоченных
целых чисел от 1 до 10
library(keras)
int_sequence <- seq(10)
dummy_dataset <- timeseries_dataset_from_array(
  data = head(int_sequence, -3),
  targets = tail(int_sequence, -3),
  sequence_length = 3,
  batch_size = 2
)
Последовательность, которую мы генерируем,
будет взята из массива [1 2 3 4 5 6 7]
(отбрасывая последние 3)
Последовательностью
целей, начинающейся
с data[N], будет data[N + 4]
(отбрасывая первые 3)
Последовательность
будет составлять три
шага в длину
Последовательности будут объединены в пакеты размером 2
library(tfdatasets)
dummy_dataset_iterator <- as_array_iterator(dummy_dataset)

repeat {
  batch <- iter_next(dummy_dataset_iterator)
  if (is.null(batch)) break
  c(inputs, targets) %<-% batch
  for (r in 1:nrow(inputs))
    cat(sprintf("input: [ %s ] target: %s\n",
               paste(inputs[r, ], collapse = " "), targets[r]))
  cat(strrep("-", 27), "\n")
}
Итератор исчерпан
Маркировка пакетов

```

Этот фрагмент кода выводит следующие результаты:

```

input: [ 1 2 3 ] target: 4
input: [ 2 3 4 ] target: 5
-----
input: [ 3 4 5 ] target: 6
input: [ 4 5 6 ] target: 7
-----
input: [ 5 6 7 ] target: 8
-----

```

Мы будем использовать `timeseries_dataset_from_array()` для создания экземпляров трех наборов данных: один для обучения, один для проверки и один контрольный. Для этого отправим в утилиту следующие аргументы:

- `sample_rate = 6` – выборка наблюдений будет производиться по одной точке данных в час: мы будем хранить только одну точку данных из 6;
- `sequence_length = 120` – наблюдения будут просматриваться на пять дней назад (120 часов);
- `delay = sampling_rate * (sequence_length + 24 - 1)` – целью обучающей последовательности будет температура через 24 часа после окончания последовательности.

### Листинг 10.6 Создание экземпляров наборов данных для обучения, проверки и контроля

```

sampling_rate <- 6
sequence_length <- 120
delay <- sampling_rate * (sequence_length + 24 - 1)
batch_size <- 256

df_to_inputs_and_targets <- function(df) {
  inputs <- df[input_data_colnames] %>%
    normalize_input_data() %>%
    as.matrix()
  targets <- as.array(df$`T (degC)` )
  list(
    head(inputs, -delay),
    tail(targets, -delay)
  )
}

make_dataset <- function(df) {
  c(inputs, targets) %<% df_to_inputs_and_targets(df)
  timeseries_dataset_from_array(
    inputs, targets,
    sampling_rate = sampling_rate,
    sequence_length = sequence_length,
    shuffle = TRUE,
    batch_size = batch_size
  )
}

train_dataset <- make_dataset(train_df)
val_dataset <- make_dataset(val_df)
test_dataset <- make_dataset(test_df)

```

Конвертируем фрейм данных в числовой массив

Мы не нормализуем цели

Отбрасываем последние выборки

Отбрасываем первые выборки

Каждый набор данных дает пакеты в виде пары (`samples`, `targets`), где `samples` – это пакет из 256 образцов, каждый из которых содержит 120 последовательных часов входных данных, а `targets` – это соответствующий массив из 256 целевых температур. Обратите внимание, что образцы перемешиваются случайным образом, поэтому две соседние последовательности в пакете (например, `samples[1, ]` и `samples[2, ]`) не обязательно близки по времени.

**Листинг 10.7 Проверка вывода для одного из наборов данных**

```
c(samples, targets) %<-% iter_next(as_iterator(train_dataset))
cat("samples shape: ", format(samples$shape), "\n",
    "targets shape: ", format(targets$shape), "\n", sep = "")
```

```
samples shape: (256, 120, 14)
targets shape: (256)
```

## 10.2.2 Простое решение задачи 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 определить базовый уровень точности, которую мы должны будем превзойти, чтобы доказать преимущество более сложных моделей машинного обучения. Такие очевидные базовые решения могут использоваться, когда вы подступаетесь к новой задаче, не имеющей (пока) известного решения. Классическим примером могут служить несбалансированные задачи классификации, когда некоторые классы могут быть намного более распространены, чем другие. Если набор данных содержит 90 % экземпляров класса «А» и 10 % экземпляров класса «Б», тогда очевидным решением задачи классификации является неизменный выбор класса «А» для предсказания классов новых образцов. Такой классификатор будет иметь общую точность 90 %, и, соответственно, любое решение на основе машинного обучения должно превзойти эти 90 %, чтобы доказать свою полезность. Иногда такие элементарные базовые решения на удивление трудно превзойти.

В данном случае временные последовательности можно с полной уверенностью считать монотонными (температура завтра, вероятно, будет близка к сегодняшней), а также подчиняющимися суточной периодичности. То есть разумным базовым прогнозом температуры через 24 часа является текущая температура. Давайте оценим этот подход, используя метрику средней абсолютной ошибки (Mean Absolute Error, MAE):

```
mean(abs(preds - targets))
```

Цикл оценки представлен в листинге 10.8. Вместо того чтобы выполнять все операции в R, используя `for`, `as_array_iterator()` и `iter_next()`, мы можем так же легко проделать это с преобразованиями TensorFlow для наборов данных. Сначала вызовем `dataset_unbatch()`, чтобы каждый элемент набора данных превратился в отдельную пару (`samples`, `target`). Затем применим `dataset_map()` для вычисления абсолютной ошибки для каждой пары (`samples`, `target`) и `dataset_reduce()` для накопления общей ошибки и общего количества просмотренных выборок.

Напомним, что функции, переданные в `dataset_map()` и `dataset_reduce()`, будут вызываться с символьными тензорами. Операция среза тензора с отрицательным числом, например `samples[-1, ]`, выбирает последний срез по этой оси, как если бы мы использовали `samples[nrow(samples), ]`.

### Листинг 10.8 Вычисление MAE без использования глубокого обучения

```
evaluate_naive_method <- function(dataset) {
  unnormalize_temperature <- function(x) {
    nv <- normalization_values$`T (degC)`
    (x * nv$sd) + nv$mean
  }

  temp_col_idx <- match("T (degC)", input_data_colnames)

  reduction <- dataset %>%
    dataset_unbatch() %>%
    dataset_map(function(samples, target) {
      last_temp_in_input <- samples[-1, temp_col_idx]
      pred <- unnormalize_temperature(last_temp_in_input)
      abs(pred - target)
    }) %>%
    dataset_reduce(
      initial_state = list(total_samples_seen = 0L,
                          total_abs_error = 0),
      reduce_func = function(state, element) {
        state$total_samples_seen %<>% `+`(1L)
        state$total_abs_error %<>% `+`(element)
        state
      }
    ) %>%
    lapply(as.numeric)

  mae <- with(reduction,
             total_abs_error / total_samples_seen)

  mae
}

sprintf("Validation MAE: %.2f", evaluate_naive_method(val_dataset))
sprintf("Test MAE: %.2f", evaluate_naive_method(test_dataset))
```

2, второй столбец

Срез последнего измерения температуры во входной последовательности

Напомним, что мы нормализовали наши признаки, поэтому, чтобы получить температуру в градусах Цельсия, нам нужно ее денормализовать, умножив на стандартное отклонение и прибавив среднее значение

Конвертация тензора в числовые значения R

reduction – это именованный список двух скаляров R

```
[1] "Validation MAE: 2.43"
[1] "Test MAE: 2.62"
```

Эта базовая оценка без использования машинного обучения обеспечивает проверочную MAE 2,44 °C и контрольную MAE 2,62 °C. Поэтому, если вы возьметесь предполагать, что температура спустя 24 часа будет такой же, как сейчас, вы ошибетесь в среднем на два с половиной градуса. Это выглядит не так уж плохо с точки зрения

статистики, но сервис прогноза погоды, основанный на подобной эвристике, скорее всего, не будет пользоваться спросом. Наша цель заключается в том, чтобы использовать свои знания в области глубокого обучения и добиться лучшего результата.

### 10.2.3 Решение с использованием базовой модели машинного обучения

Прежде чем пытаться создать такую сложную и затратную (в вычислительном смысле) модель, как рекуррентная нейронная сеть, помимо базового решения без привлечения машинного обучения также полезно испытать простые и незатратные модели машинного обучения (например, неглубокую полносвязную сеть). Это лучший способ убедиться, что любые усложнения, направленные на решение задачи, оправданны и действительно дают преимущества.

В листинге 10.9 показана полносвязная модель, которая начинается со снижения размерности данных, а затем проходит через два слоя `layer_dense()`. Обратите внимание на отсутствие функции активации в последнем слое `layer_dense()`, что типично для задачи регрессии. Мы используем в качестве метрики потерь среднеквадратичную ошибку (MSE), а не MAE, потому что, в отличие от MAE, она гладкая около нуля, что является полезным свойством для градиентного спуска. Мы будем отслеживать MAE, добавляя ее в качестве метрики в `compile()`.

#### Листинг 10.9 Обучение и оценка полносвязной модели

```
ncol_input_data <- length(input_data_colnames)

inputs <- layer_input(shape = c(sequence_length, ncol_input_data))
outputs <- inputs %>%
  layer_flatten() %>%
  layer_dense(16, activation = "relu") %>%
  layer_dense(1)
model <- keras_model(inputs, outputs)

callbacks = list(
  callback_model_checkpoint("jena_dense.keras", ←
    save_best_only = TRUE)
)

model %>%
  compile(optimizer = "rmsprop",
    loss = "mse",
    metrics = "mae")

history <- model %>%
  fit(train_dataset,
    epochs = 10,
```

Используем обратный вызов для сохранения наилучшей модели

```
validation_data = val_dataset,
callbacks = callbacks)

model <- load_model_tf("jena_dense.keras")
sprintf("Test MAE: %.2f", evaluate(model, test_dataset)["mae"])
```

Загружаем сохраненную  
лучшую модель и оцениваем  
ее на контрольных данных

```
[1] "Test MAE: 2.71"
```

Построим кривые потерь для проверки и обучения (рис. 10.3).

#### Листинг 10.10 Построение графиков потерь

```
plot(history, metrics = "mae")
```

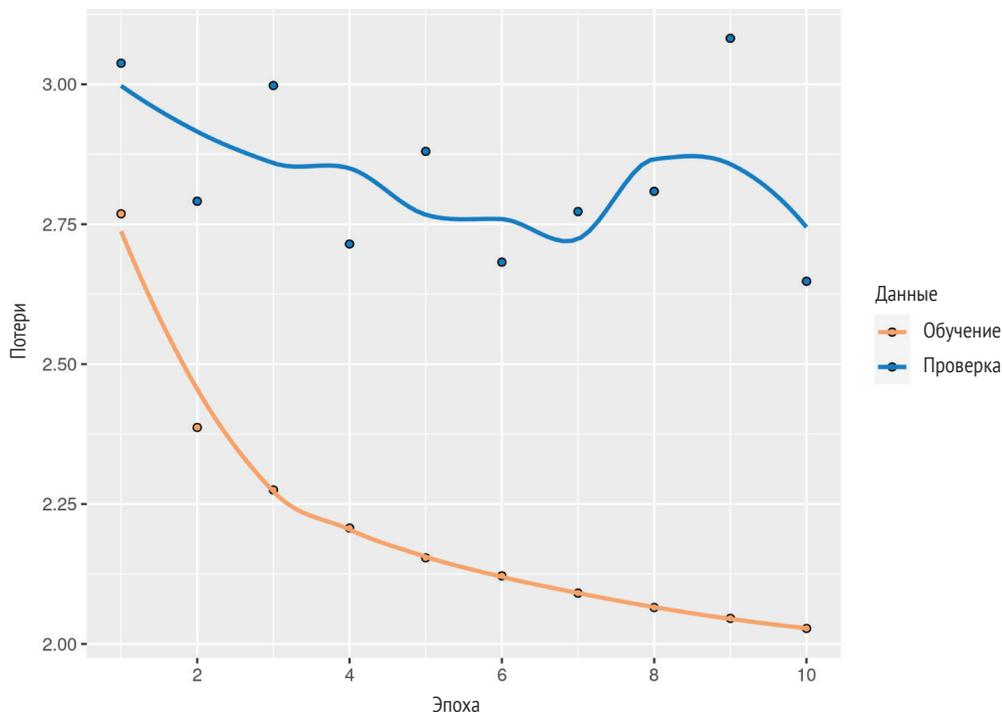


Рис. 10.3 Проверка MAE на задаче прогнозирования температуры в Йене с простой полносвязной сетью

Некоторые значения потерь на этапе проверки близки к оценке ошибки базового решения без привлечения машинного обучения, но эта ситуация нестабильна. 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: раз существует хорошая и простая модель прогнозирования целей по имеющимся данным (базовое решение), почему обучаемая модель не смогла показать лучшие ре-

зультаты? Потому что это простое решение совсем не то, что пытаются найти обучаемая модель. Пространство моделей, в котором мы ищем решение, то есть пространство гипотез, – это пространство всех возможных двухслойных сетей с определяемой нами конфигурацией. Эти сети уже довольно сложны. Когда поиск решения выполняется в пространстве сложных моделей, простое базовое решение может оказаться недостижимым, даже если технически оно является частью пространства гипотез.

Это существенное ограничение машинного обучения в целом: если алгоритм обучения не запрограммирован на поиск конкретной простой модели, обучение параметров иногда может терпеть неудачу в попытках найти простое решение простой задачи. Вот почему важно использовать качественное конструирование признаков и подходящие архитектуры, основанные на априорном знании задачи: иными словами, нужно точнее указать своей модели, что она должна искать.

### 10.2.4 Эксперимент с одномерной сверточной сетью

Я неоднократно упоминал о выборе правильной априорно обусловленной архитектуры. Поскольку наши входные последовательности имеют ежедневные циклы, можно предположить, что хорошо работает сверточная сеть. Свертка по времени должна позволить повторно использовать одни и те же представления в разные дни, так же как пространственная свертка позволяет повторно использовать одни и те же представления в разных местах изображения.

Вы уже знаете о `layer_conv_2d()` и `layer_separable_conv_2d()`, которые видят свои входные данные через маленькие окна, перемещаемые по 2D-сеткам. Существуют также одномерные и даже трехмерные версии этих слоев: `layer_conv_1d()`, `layer_separable_conv_1d()` и `layer_conv_3d()`<sup>1</sup>. Слой `layer_conv_1d()` основан на одномерных окнах, которые скользят по входным последовательностям, а слой `layer_conv_3d()` основан на кубических окнах, которые скользят по входным объемам.

Следовательно, вы можете строить одномерные сверточные сети, строго аналогичные двумерным сверточным сетям. Они отлично подходят для любых данных в последовательности, которые следу-

---

<sup>1</sup> Слой `layer_separable_conv_3d()` отсутствует не по какой-либо теоретически обоснованной причине, а просто потому, что я не реализовал его.

ют предположению об инвариантности трансляции (это означает, что если вы перемещаете окно по последовательности, содержимое окна должно обладать одинаковыми свойствами независимо от местоположения окна).

Давайте попробуем одну из таких сетей на нашей задаче прогнозирования температуры. Мы выберем начальную длину окна 24, чтобы просматривать данные за 24 часа за один раз (один цикл). По мере того как мы понижаем разрешение последовательностей (через слои `layer_max_pooling_1d()`), мы соответственно уменьшаем размер окна:

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))
outputs <- inputs %>%
  layer_conv_1d(8, 24, activation = "relu") %>%
  layer_max_pooling_1d(2) %>%
  layer_conv_1d(8, 12, activation = "relu") %>%
  layer_max_pooling_1d(2) %>%
  layer_conv_1d(8, 6, activation = "relu") %>%
  layer_global_average_pooling_1d() %>%
  layer_dense(1)
model <- keras_model(inputs, outputs)

callbacks <- list(callback_model_checkpoint("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 <- load_model_tf("jena_conv.keras")
sprintf("Test MAE: %.2f", evaluate(model, test_dataset)["mae"])

[1] "Test MAE: 3.20"
```

Мы получаем кривые обучения и проверки, показанные на рис. 10.4.

```
plot(history)
```

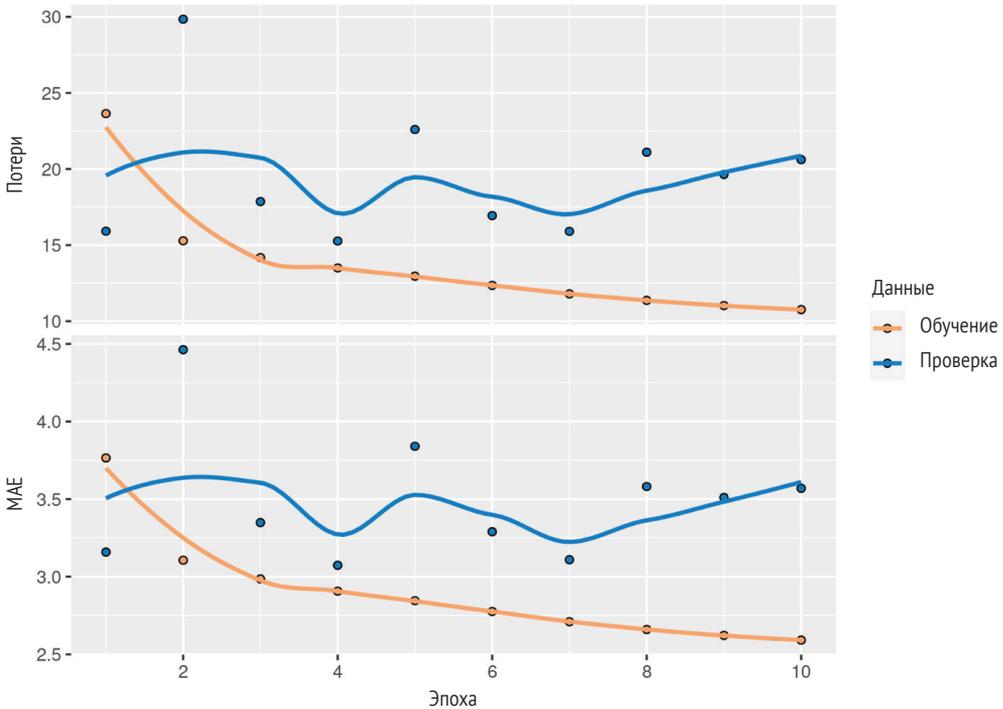


Рис. 10.4 Обучение и проверка MAE для задачи прогнозирования температуры в Йене с одномерной сверточной сетью

Как оказалось, эта модель работает даже хуже, чем простая полностью связная модель, выдавая значение MAE на контрольных данных 3,2 °C, что далеко от уровня базовой модели без использования глубокого обучения. Что здесь пошло не так? Две вещи:

- во-первых, данные о погоде не совсем соответствуют предположению о трансляционной инвариантности. Хотя данные содержат суточные циклы, утренние данные обладают другими свойствами, чем вечерние или ночные данные. Данные о погоде инвариантны к трансляции только для очень ограниченного временного масштаба;
- во-вторых, порядок в наших данных имеет большое значение. Недавнее прошлое гораздо более информативно для предсказания температуры следующего дня, чем данные пятидневной давности. Одномерная сеть не может использовать этот факт. В частности, наши слои объединения по максимальному соседнему и глобального объединения по средним значениям почти полностью уничтожают информацию, заключенную в порядке измерений.

## 10.2.5 Первый вариант простой рекуррентной модели

Ни полносвязная, ни одномерная сверточная модель не сработали хорошо, но это не значит, что машинное обучение неприменимо к задаче предсказания временных рядов. Полносвязная модель начинала с удаления понятия времени из входных данных. Сверточная модель обрабатывала каждый сегмент данных одинаково, но применяла объединение, которое уничтожало информацию о порядке. Давайте попробуем рассмотреть данные как последовательность, в которой важны причинно-следственная связь и порядок.

Существует семейство архитектур нейронных сетей, разработанных специально для данных такого рода: *рекуррентные нейронные сети*. Среди них уже давно очень популярна архитектура на основе ячеек *долгой краткосрочной памяти* (long short-term memory, LSTM). Скоро вы узнаете, как работают эти модели, но давайте начнем с того, что попробуем в деле один слой LSTM.

### Листинг 10.11 Простая модель на основе LSTM

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))
outputs <- inputs %>%
  layer_lstm(16) %>%
  layer_dense(1)
model <- keras_model(inputs, outputs)

callbacks <- list(callback_model_checkpoint("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 <- load_model_tf("jena_lstm.keras")
sprintf("Test MAE: %.2f", evaluate(model, test_dataset)["mae"])

[1] "Test MAE: 2.52"
```

На рис. 10.5 показаны результаты. Намного лучше! Мы достигли контрольного значения MAE 2,52 °C. Модель на основе LSTM, наконец, смогла превзойти базовый уровень модели без глубокого обучения (хотя пока незначительно), демонстрируя ценность машинного обучения для этой задачи.

Но почему модель LSTM работает заметно лучше, чем полносвязная или сверточная сеть? И как еще можно улучшить модель? Чтобы ответить на этот вопрос, давайте детально рассмотрим принцип работы рекуррентных сетей.

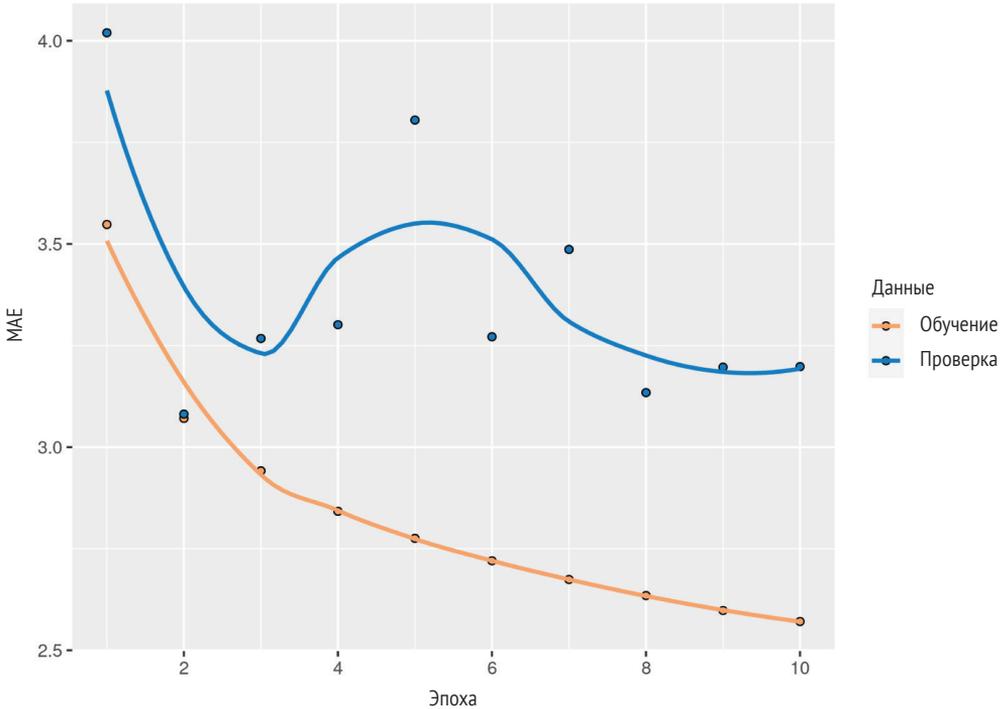


Рис. 10.5 Обучение и проверка MAE в задаче прогнозирования температуры в Йене с моделью на основе LSTM (обратите внимание, что мы опускаем эпоху 1 на этом графике, потому что высокое значение MAE в первой эпохе исказит масштаб)

### 10.3 Рекуррентные нейронные сети

Ключевая общая черта полносвязных и сверточных сетей, рассмотренных в предыдущих главах, заключается в том, что у них нет памяти. Эти сети обрабатывают каждый вход заново, при этом между входами не сохраняется состояние. В таких сетях, чтобы обработать временной ряд точек данных, вы должны показать сети сразу всю последовательность, фактически превратив ее в единую точку данных. Именно это мы сделали в примере с плотной сетью: объединили данные за пять дней в один большой вектор и обработали его за один раз. Такие сети называются *сетями прямого распространения*.

Наш мозг в этом смысле работает иначе. Например, когда вы читаете данное предложение, вы обрабатываете его слово за сло-

вом – или, скорее, одну *саккаду* глаз за другой, – сохраняя при этом воспоминания о том, что было раньше; это дает вам цельное представление смысла, заложенного в это предложение. Биологический интеллект обрабатывает информацию постепенно, сохраняя при этом внутреннюю модель того, что он обрабатывает, построенную на основе прошлой информации и постоянно обновляемую по мере поступления новых данных.

*Рекуррентная нейронная сеть* (recurrent neural network, RNN) использует тот же принцип, хотя и в чрезвычайно упрощенном виде: она перебирает элементы последовательности и поддерживает *состояние*, сохраняющее информацию о том, что она видела до сих пор. По сути, RNN – это разновидность нейронной сети с внутренним циклом (рис. 10.6).

Состояние RNN сбрасывается между обработкой двух разных независимых последовательностей (например, двух выборок в пакете), поэтому одну последовательность по-прежнему можно считать одной точкой данных или одним вводом в сеть. Принципиально изменился единственный, но очень важный аспект: точка данных больше не обрабатывается за один шаг. Внутри RNN происходит циклический перебор элементов входной последовательности.

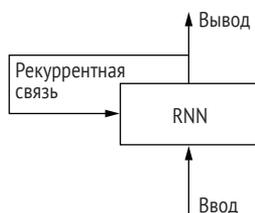


Рис. 10.6 Внутренний цикл рекуррентной нейронной сети

Чтобы прояснить понятия *цикла* и *состояния*, реализуем прямой проход демонстрационной RNN. Эта RNN принимает в качестве входных данных последовательность векторов, которые мы будем кодировать как тензор второго ранга с размером (`timesteps`, `input_features`). Она циклически обрабатывает шаги времени (такты) и на каждом шаге рассматривает свое текущее состояние в момент  $t$  и ввод (`input_features`) в этот момент и объединяет их для получения вывода в момент времени  $t$ . Далее этот вывод вместе со следующим элементом последовательности поступает на вход следующего шага. Для первого шага предыдущий вывод не определен; следовательно, нет текущего состояния. Поэтому мы инициализируем состояние как нулевой вектор, называемый *начальным состоянием* сети. В листинге 10.12 показан псевдокод такой RNN.

#### Листинг 10.12 Псевдокод RNN

```
state_t <- 0  ← Состояние в момент t
for (input_t in input_sequence) { ← Итерация по элементам
                                последовательности
```

```

output_t <- f(input_t, state_t)
state_t <- output_t
}

```

← Предыдущий выход становится состоянием для следующей итерации

Вы даже можете конкретизировать функцию  $f$ : преобразование входа и состояния в выход будет параметризовано двумя матрицами,  $W$  и  $U$ , и вектором смещения. Это похоже на преобразование, выполняемое полносвязным слоем в сети с прямым распространением.

### Листинг 10.13 Более подробный псевдокод для RNN

```

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 закрепить понимание, давайте напишем на R базовую реализацию прямого прохода простой RNN.

### Листинг 10.14 Базовая реализация прямого прохода простой RNN на R

```

random_array <- function(dim) array(runif(prod(dim)), dim)

timesteps <- 100
input_features <- 32
output_features <- 64

inputs <- random_array(c(timesteps, input_features))
state_t <- array(0, dim = output_features)

W <- random_array(c(output_features, input_features))
U <- random_array(c(output_features, output_features))
b <- random_array(c(output_features, 1))
successive_outputs <- array(0, dim = c(timesteps, output_features))

for (ts in 1:timesteps) {
  input_t <- inputs[ts, ]
  output_t <- tanh((W %%% input_t) + (U %%% state_t) + b)
  successive_outputs[ts, ] <- output_t
  state_t <- output_t
}
final_output_sequence <- successive_outputs

```

← Количество временных шагов во входной последовательности

← Размерность входного пространства признаков

← Размерность выходного пространства признаков

← Входные данные: случайный шум для примера

← Создание матрицы случайных весов

←  $input\_t$  – вектор формы ( $input\_features$ )

← Сохранение данного вывода

← Обновление состояния сети для следующего временного шага

Исходное состояние: нулевой вектор

Конечным результатом является тензор 2 ранга формы ( $timesteps, output\_features$ )

$W \% \% input\_t, U \% \% input\_t$  и  $b$  имеют одинаковую форму: ( $output\_features, 1$ )

Объединение ввода с текущим состоянием (предыдущий вывод), чтобы получить текущий вывод. Мы используем  $\tanh$  для добавления нелинейности (мы могли бы использовать любую другую функцию активации)

Как видите, в этом коде нет ничего сложного. Фактически RNN – это цикл `for`, который повторно использует значения, вычисленные во время предыдущей итерации цикла, не более того. Конечно, вы можете построить множество различных RNN, соответствующих этому определению, – этот пример представляет собой одну из самых простых архитектур RNN. Рекуррентная сеть определяется *ступенчатой функцией* (рис. 10.7); в данном случае это функция следующего вида:

```
output_t <- tanh((W %% input_t) + (U %% state_t) + b)
```

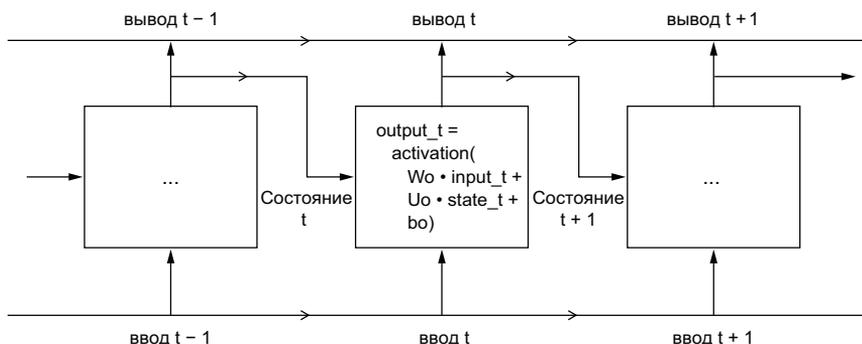


Рис. 10.7 Простая RNN, развернутая во времени

В этом примере окончательный результат – это тензор второго ранга формы  $(\text{timesteps}, \text{output\_features})$ , где каждый временной шаг – это выходные данные цикла в момент времени  $t$ . Каждый шаг  $t$  в выходном тензоре содержит информацию о временных шагах с 1 по  $t$  во входной последовательности – весь прошедший период. По этой причине во многих случаях вам не нужна полная последовательность выходных данных; достаточно последнего вывода ( $\text{output}_t$  в конце цикла), потому что он уже содержит информацию обо всей последовательности.

### 10.3.1 Рекуррентный слой в Keras

Процесс, который мы только что реализовали на языке R, соответствует реальному слою Keras – `layer_simple_rnn()`. Есть одно небольшое отличие: `layer_simple_rnn()` обрабатывает пакеты последовательностей, как и все остальные слои Keras, а не одну последовательность, как в примере с R. Это означает, что он принимает входные данные формы  $(\text{batch\_size}, \text{timesteps}, \text{input\_features})$ , а не  $(\text{timesteps}, \text{input\_features})$ . В аргументе формы начальных входных данных вы можете установить для записи временных шагов значение `NA`, что позволит вашей сети обрабатывать последовательности произвольной длины.

### Листинг 10.15 Слой RNN для обработки последовательности произвольной длины

```
num_features <- 14
inputs <- layer_input(shape = c(NA, num_features))
outputs <- inputs %>% layer_simple_rnn(16)
```

Это особенно полезно, если ваша модель предназначена для обработки последовательностей переменной длины. Однако если все ваши последовательности гарантированно имеют одинаковую длину, я рекомендую полностью определять форму ввода, потому что это позволяет методу `print()` модели отображать информацию о длине вывода, что всегда удобно, и открывает дополнительные возможности для оптимизации производительности (см. врезку в разделе 10.4.1).

Все рекуррентные слои в Keras (`layer_simple_rnn()`, `layer_lstm()` и `layer_gru()`) могут выполняться в двух разных режимах: они могут возвращать или полные последовательности выходных данных для каждого временного шага (трехмерный тензор формы `(batch_size, timesteps, output_features)`), или только последний вывод для каждой входной последовательности (двухмерный тензор формы `(batch_size, output_features)`). За выбор режима отвечает аргумент `return_sequences`. Рассмотрим пример в листинге 10.16, который использует `layer_simple_rnn()` и возвращает только вывод на последнем временном шаге.

### Листинг 10.16 Уровень RNN, возвращающий только последний выходной шаг

```
num_features <- 14
steps <- 120
inputs <- layer_input(shape = c(steps, num_features))
outputs <- inputs %>%
  layer_simple_rnn(16, return_sequences = FALSE)
outputs$shape
TensorShape([None, 16])
```

Помните, что по умолчанию  
return\_sequences = FALSE

В следующем примере (листинг 10.17) возвращается полная последовательность состояний.

### Листинг 10.17 Уровень RNN, возвращающий полную выходную последовательность

```
num_features <- 14
steps <- 120
inputs <- layer_input(shape = c(steps, num_features))
outputs <- inputs %>% layer_simple_rnn(16, return_sequences = TRUE)
outputs$shape
TensorShape([None, 120, 16])
```

Иногда полезно расположить несколько рекуррентных слоев один за другим, чтобы увеличить репрезентативную способность сети (листинг 10.18). В такой архитектуре необходимо заставить все промежуточные слои возвращать полную последовательность выходных данных.

#### Листинг 10.18 Последовательное расположение нескольких слоев RNN

```
inputs <- layer_input(shape = c(steps, num_features))
outputs <- inputs %>%
  layer_simple_rnn(16, return_sequences = TRUE) %>%
  layer_simple_rnn(16, return_sequences = TRUE) %>%
  layer_simple_rnn(16)
```

На практике вы редко будете работать с `layer_simple_rnn()`. Как правило, этот слой слишком прост, чтобы иметь реальное применение. В частности, у `layer_simple_rnn()` есть серьезная проблема: хотя теоретически он в состоянии сохранить в момент времени  $t$  информацию о входных данных, которые были просмотрены за много временных шагов ранее, такие долгосрочные зависимости невозможно изучить на практике. Это происходит из-за проблемы исчезающего градиента, эффекта, похожего на то, что наблюдается в многослойных сетях с прямым распространением: по мере того как вы продолжаете добавлять слои в сеть, она теряет способность к обучению. Теоретические причины этого эффекта изучены Хохрайтером, Шмидхубером и Бенжио в начале 1990-х годов<sup>1</sup>.

К счастью, `layer_simple_rnn()` – не единственный рекуррентный слой, доступный в Keras. Есть два других, `layer_lstm()` и `layer_gru()`, специально разработанных для решения проблемы исчезающего градиента.

Начнем со слоя `layer_lstm()`. Лежащий в его основе алгоритм долгой краткосрочной памяти (LSTM) был разработан Хохрайтером и Шмидхубером в 1997 г.<sup>2</sup> Он стал кульминацией их исследования проблемы исчезающего градиента.

Фактически это усовершенствованный вариант слоя `layer_simple_rnn()`, о котором вы уже знаете; в него добавлен способ переноса информации через множество временных шагов. Представьте конвейерную ленту, идущую параллельно последовательности, которую вы обрабатываете. Информация из последовательности может переместиться с лентой конвейера на более поздний временной шаг и поступить в сеть в неизменном виде, когда она вам понадобится.

<sup>1</sup> Например, Yoshua Bengio, Patrice Simard, Paolo Frasconi, *Learning Long-Term Dependencies with Gradient Descent Is Difficult*, IEEE Transactions on Neural Networks 5, no. 2 (1994).

<sup>2</sup> Sepp Hochreiter, Jürgen Schmidhuber, *Long Short-Term Memory*, Neural Computation 9, no. 8 (1997).

Именно это и делает слой LSTM: он сохраняет информацию «на потом», тем самым предотвращая постепенное угасание старых сигналов во время обработки. Наверняка вы вспомнили о сети с остаточной связью из главы 9: это очень похожая идея.

Чтобы лучше понять рабочий процесс, начнем с ячейки `layer_simple_rnn()` (рис. 10.8). Поскольку у нас будет много весовых матриц, пометим матрицы  $W$  и  $U$  в ячейке буквой  $o$  ( $W_o$  и  $U_o$ ) от слова `output` – вывод.

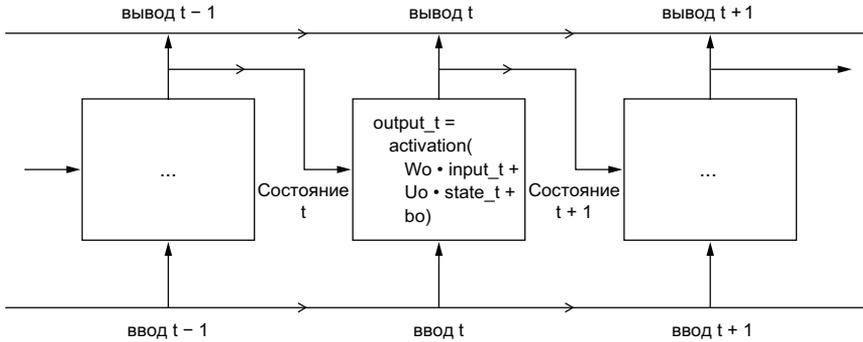


Рис. 10.8 Исходная точка слоя LSTM: SimpleRNN

Теперь добавим к этой схеме дополнительный канал данных для переноса информации между шагами. Обозначим его значения на разных временных шагах как  $c_t$ , где  $c$  означает carry (перенос). Эта информация складывается с входной информацией ячейки (путем скалярного перемножения с матрицей весов, за которым следует добавление смещения и применение функции активации) и тем самым влияет на состояние, отправляемое на следующий временной шаг. С технической точки зрения, перенос информации в специальном канале – это способ модуляции следующего вывода и следующего состояния (рис. 10.9). Пока ничего сложного.

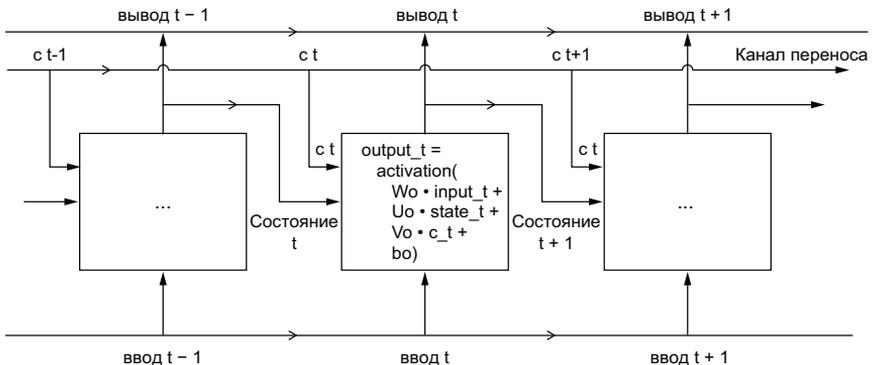


Рис. 10.9 Переход от SimpleRNN к LSTM: добавление канала переноса



канал переноса новой информацией. Но, в конце концов, эти мысленные толкования мало что значат, потому что фактическая роль этих операций определяется их собственными параметризующими весами; а веса, как вы догадываетесь, формируются сквозным образом в течение нескольких эпох обучения. Поэтому невозможно приписать той или иной операции конкретную роль. Спецификация ячейки RNN определяет пространство вашей гипотезы – пространство, в котором вы будете искать хорошую конфигурацию модели во время обучения, – но не определяет, что именно делает ячейка; это зависит от весов ячейки. Одна и та же ячейка с разными наборами весовых коэффициентов может делать совершенно разные вещи. Следовательно, комбинацию операций, составляющих ячейку RNN, лучше интерпретировать как *набор ограничений* на ваш поиск, а не как *конструкцию* в инженерном смысле.

Возможно, выбор таких ограничений – вопрос о том, как реализовать ячейки RNN, – лучше предоставить алгоритмам оптимизации (например, генетическим алгоритмам или процессам обучения с подкреплением), чем разработчикам-людям. В будущем именно так мы будем строить наши модели. Иными словами, вам не нужно вникать в детали устройства каждой ячейки LSTM; это не та работа, которую должен делать человек. Вам достаточно понимать назначение ячейки LSTM: повторный ввод прежней информации в более позднее время для борьбы с проблемой исчезающего градиента.

## 10.4 Продвинутое применение рекуррентных нейронных сетей

К этому моменту вы узнали следующее:

- что такое RNN и как они работают;
- что такое сети LSTM и почему они лучше работают с длинными последовательностями, чем простые RNN;
- как использовать слои Keras RNN для обработки последовательностей данных.

Далее мы рассмотрим более продвинутые функциональные возможности RNN, опираясь на которые, вы сможете получить максимальную отдачу от ваших рекуррентных моделей глубокого обучения. В этом разделе вы освоите основные навыки использования рекуррентных сетей на платформе Keras.

Итак, в заключительной части главы мы рассмотрим следующие темы:

- *рекуррентное прореживание* (recurrent dropout) – это вариант прореживания, используемый для борьбы с переобучением в рекуррентных слоях;

- *наложение рекуррентных слоев* – увеличение репрезентативной способности модели (ценой более высоких вычислительных нагрузок);
- *двунаправленные рекуррентные сети* – способ представить одну и ту же информацию в рекуррентной сети по-разному для повышения точности и решения проблемы забывания.

Для демонстрации этих методов мы воспользуемся уже знакомой RNN прогнозирования температуры.

### 10.4.1 Использование рекуррентного прореживания для борьбы с переобучением

Давайте вернемся к модели на основе LSTM, которую мы использовали в разделе 10.2.5, – нашей первой модели, способной превзойти базовый уровень простой модели без глубокого обучения. Из кривых потерь на этапах обучения и проверки видно, что в модели быстро наступает эффект переобучения: потери начинают значительно отличаться 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 уровень. Но правильное применение прореживания в рекуррентных сетях – нетривиальная задача.

Давно известно, что применение прореживания перед рекуррентным уровнем скорее мешает обучению, чем помогает регуляризации. В 2015 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 временным интервалам должна применяться одна и та же маска прореживания (должны обнуляться одни и те же значения), не изменяющаяся от интервала к интервалу. Более того, для регуляризации представлений, сформированных рекуррентными уровнями, такими как `layer_glu` и `layer_lstm`, временно-постоянная маска прореживания должна применяться к внутренним рекуррентным активациям уровня (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 уровни Keras. Каждый рекуррентный слой в Keras имеет два аргумента,

<sup>1</sup> Yarin Gal, *Uncertainty in Deep Learning*, PhD thesis (2016), <http://mng.bz/WBq1>.

связанных с прореживанием: `dropout`, вещественное число, определяющее долю прореживаемых входных значений уровня, и `recurrent_dropout`, определяющий долю прореживаемых рекуррентных значений. Давайте добавим прореживание входных и рекуррентных значений в уровень `layer_lstm()` и посмотрим, как это повлияет на переобучение.

Благодаря прореживанию нам не нужно сильно беспокоиться о размере сети для регуляризации, поэтому мы будем использовать слой LSTM с вдвое большим количеством элементов, который, как мы надеемся, должен быть более выразительным (без прореживания эта сеть сразу начинает переобучаться – попробуйте сами). Поскольку для полной сходимости сетей, регуляризованных с применением прореживания, всегда требуется гораздо больше времени, мы увеличим в пять раз количество эпох обучения по сравнению с предыдущим примером.

#### Листинг 10.21 Обучение и оценка модели LSTM с регуляризацией прореживанием

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))
outputs <- inputs %>%
  layer_lstm(32, recurrent_dropout = 0.25) %>%
  layer_dropout(0.5) %>% ←
  layer_dense(1)
model <- keras_model(inputs, outputs)

callbacks <- list(callback_model_checkpoint("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
}
```

Для регуляризации полносвязного слоя мы также добавляем слой прореживания после LSTM

Результаты показаны на рис. 10.11. Это успех! Теперь эффект переобучения не наблюдается на протяжении первых 15 эпох. На проверочных данных мы получили MAE 2,37 °C (улучшение точности на 2,5 % по сравнению с базовой моделью без глубокого обучения) и на контрольных данных MAE 2,45 °C (улучшение на 6,5 %). Весьма неплохо.

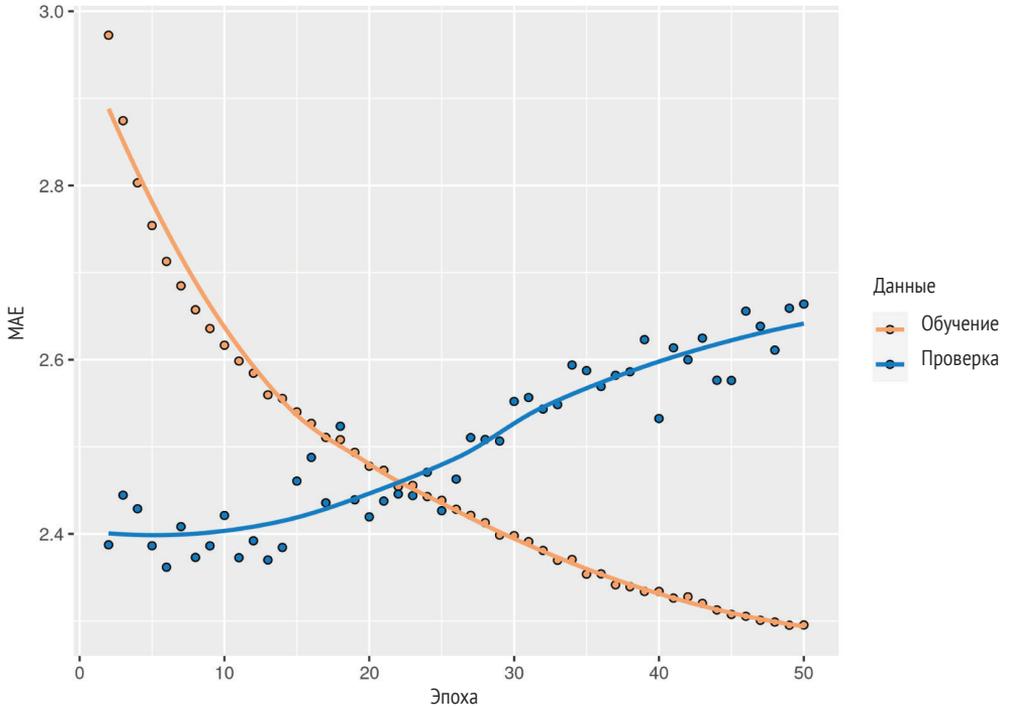


Рис. 10.11 Кривые потерь при обучении и проверке для задачи прогнозирования температуры в Йене с LSTM после регуляризации прореживанием

### Повышение быстродействия RNN

Рекуррентные модели с очень небольшим количеством параметров, такие как модели в этой главе, как правило, работают значительно быстрее на многоядерном процессоре, чем на графическом процессоре, потому что они выполняют только небольшие матричные умножения, а цепочка умножений плохо поддается распараллеливанию из-за наличия петли. Но более крупные RNN могут значительно выиграть от выполнения на GPU.

При вычислении слоя Keras LSTM или GRU на графическом процессоре с аргументами по умолчанию этот слой будет использовать ядро cuDNN – высокооптимизированную низкоуровневую реализацию базового алгоритма, предоставленную NVIDIA, о которой я упоминал в предыдущей главе. К сожалению, ядра cuDNN хоть и быстрые, но не отличаются высокой гибкостью. Если вы попытаетесь сделать что-то, что не поддерживается ядром по умолчанию, вы столкнетесь с резким замедлением модели и будете вынуждены ограничиться возможностями, которые предоставляет NVIDIA.

Например, рекуррентное прореживание не поддерживается ядрами LSTM и GRU cuDNN, поэтому добавление его к вашим слоям заставляет среду выполнения вернуться к обычной реализации TensorFlow, которая обычно работает на GPU в два–пять раз медленнее (даже несмотря на одинаковую вычислительную нагрузку).

Чтобы ускорить работу RNN, когда не получается использовать cuDNN, попробуйте выполнить *развертывание* (unrolling) слоя. Развертывание цикла `for` заключается в удалении цикла и простом повторении тела цикла  $N$  раз. Применительно к RNN развертывание позволит TensorFlow оптимизировать базовый граф вычислений, однако также значительно увеличит потребность в памяти. Поэтому развертывание подходит только для относительно небольших последовательностей (около 100 шагов или меньше). Кроме того, вы можете развернуть слой только в том случае, если количество временных шагов в данных заранее известно модели (то есть если вы передаете форму без использования записей `NA` в начальном `layer_input()`). Развертывание работает следующим образом:

```
sequence_length не может быть NA
└───┬───┘
    |   |
    |   v
inputs <- layer_input(shape = c(sequence_length, num_features))
x <- inputs %>%
  layer_lstm(32, recurrent_dropout = 0.2, unroll = TRUE)
```

← Передаем unroll = TRUE,  
чтобы разрешить развертывание

## 10.4.2 Наложение рекуррентных слоев

Избавившись от эффекта переобучения, мы столкнулись с проблемой невысокого качества, поэтому теперь нужно подумать об увеличении емкости и выразительной мощности сети. Вспомните описание обобщенного процесса машинного обучения: рекомендуется всегда стараться увеличивать емкость сети, пока на первое место не выйдет проблема переобучения (при условии что предприняты все основные меры против нее, такие как прореживание). Пока проблема переобучения не стоит остро, вероятно, сеть имеет недостаточную емкость.

Увеличение емкости сети обычно осуществляется за счет увеличения числа параметров слоя или добавления дополнительных слоев. Наложение рекуррентных слоев друг на друга – классический способ конструирования более мощных рекуррентных сетей: например, не так давно алгоритм Google Translate был основан на стеке 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. Как вы уже знаете, это делается установкой параметра `return_sequences = TRUE`.

В следующем примере мы попробуем создать стек из двух рекуррентных слоев с прореживанием. Для разнообразия мы будем использовать слои Gated Recurrent Unit (GRU) вместо LSTM. Сеть GRU очень похожа на LSTM – вы можете рассматривать ее как несколько более простую, оптимизированную версию архитектуры LSTM. Она появилась в 2014 году, когда интерес к рекуррентным сетям только начинал возрождаться в тогдашнем крошечном исследовательском сообществе<sup>1</sup>.

#### Листинг 10.22 Обучение и оценка многослойной модели GRU с регуляризацией прореживанием

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))
outputs <- inputs %>%
  layer_gru(32, recurrent_dropout = 0.5, return_sequences = TRUE) %>%
  layer_gru(32, recurrent_dropout = 0.5) %>%
  layer_dropout(0.5) %>%
  layer_dense(1)
model <- keras_model(inputs, outputs)

callbacks <- list(
  callback_model_checkpoint("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 <- load_model_tf("jena_stacked_gru_dropout.keras")
sprintf("Test MAE: %.2f", evaluate(model, test_dataset)["mae"])
```

```
[1] "Test MAE: 2.42"
```

На рис. 10.12 показаны результаты. Мы достигли тестовой MAE 2,42 °C (улучшение на 7,6 % по сравнению с исходным уровнем). Вы можете видеть, что добавленный слой немного улучшает результаты, хотя и не кардинально. На данный момент вы можете наблюдать уменьшение отдачи от увеличения емкости сети.

<sup>1</sup> Cho et al., *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches* (2014), <https://arxiv.org/abs/1409.1259>.

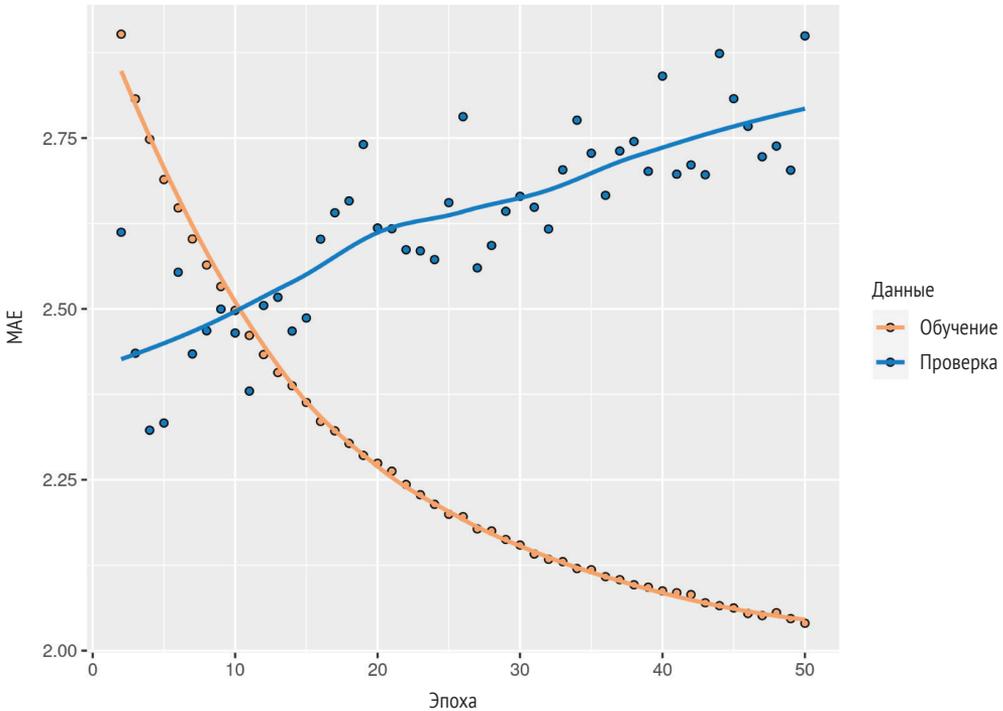


Рис. 10.12 Потери при обучении и валидации задачи прогнозирования температуры в Йене с многоуровневой сетью GRU

### 10.4.3 Использование двунаправленных рекуррентных сетей

Последний подход, который мы рассмотрим в этом разделе, называется *двунаправленная рекуррентная нейронная сеть*. Двунаправленная RNN – это распространенный вариант архитектуры, способный обеспечить более высокое качество модели для решения определенных задач. Такая сеть часто используется в обработке естественного языка – ее можно 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 `layer_gru` и `layer_lstm`, с которыми вы уже знакомы, каждая из этих сетей обрабатывает входную последовательность в одном направлении (прямом или обратном), и затем полученные представления объединяются. Об-

рабатывая последовательность в двух направлениях, двунаправленная рекуррентная сеть способна выявить шаблоны, незаметные для однонаправленной сети.

Примечательно, что хронологический порядок обработки последовательностей (от старых к новым) в этом разделе был выбран совершенно произвольно. По крайней мере, мы не пытались поставить это решение под вопрос. Могут ли рекуррентные сети показывать хорошие результаты, обрабатывая последовательности, например, в обратном порядке (от новых к старым)? Давайте попробуем применить это решение и посмотрим, что из этого получится. Для этого нужно лишь изменить набор данных TensorFlow таким образом, чтобы входные последовательности располагались в обратном порядке по оси времени. Преобразуем набор данных с помощью `dataset_map()`, как показано ниже:

```
ds %>%  
  dataset_map(function(samples, targets) {  
    list(samples[, NA:NA:-1, ], targets)  
  })
```

Обучив ту же модель на основе LSTM, которую вы использовали в первом эксперименте в этом разделе, вы получите результаты, показанные на рис. 10.13.

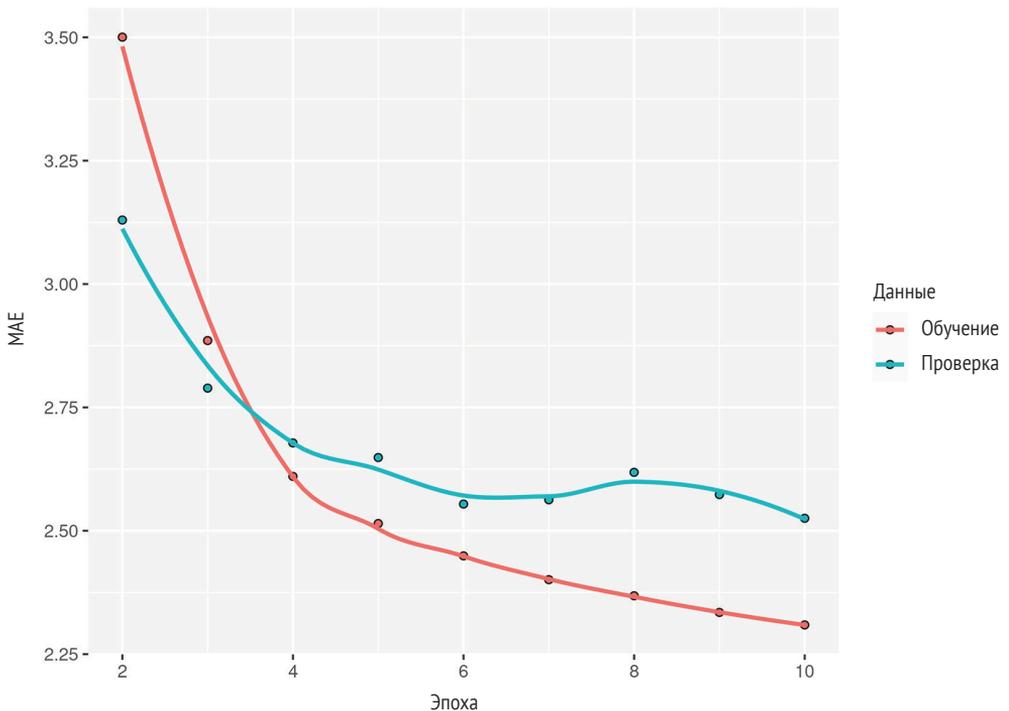


Рис. 10.13 Потери при обучении и проверке в задаче прогнозирования температуры в Йене с моделью LSTM, обученной на обратных последовательностях

Обращенная LSTM сильно уступает даже базовой модели без глубокого обучения, а это означает, что в данном случае хронологический порядок обработки последовательности принципиально важен для успешного прогнозирования. Причина достаточно очевидна: базовый слой LSTM обычно лучше запоминает недавнее прошлое, чем отдаленное, и, естественно, более свежие точки данных о погоде более предсказуемы (вот что делает простую базовую модель такой сильной). Следовательно, прямая хронологическая версия слоя должна работать лучше, чем обратная.

Однако это не всегда верно для многих других задач, включая естественный язык: очевидно, важность слова для понимания предложения обычно не зависит от его позиции в этом предложении. В случае текстовых данных обработка в обратном порядке работает так же хорошо, как и в прямом, – вы можете прекрасно читать текст в обратном порядке и понимать его смысл (попробуйте!). Это подтверждает гипотезу о том, что хотя порядок следования слов в предложении важен для его понимания, направление обработки предложений не имеет решающего значения. Следует также отметить, что рекуррентная сеть, обученная на обращенных последовательностях, получит иные представления, так же как вы сами получили бы разные ментальные модели, если бы время текло в обратном направлении и вы проживали свою жизнь в направлении от смерти к рождению. В машинном обучении не следует пренебрегать разными, но полезными представлениями, и чем больше они отличаются, 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.

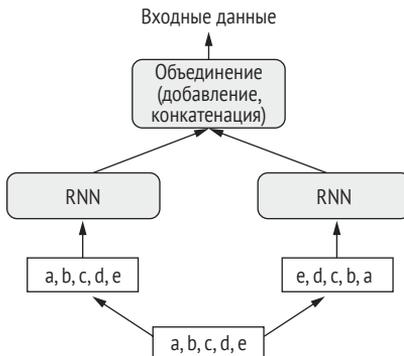


Рис. 10.14 Как работает слой двунаправленной RNN

Для создания двунаправленной рекуррентной сети в Keras имеется слой `bidirectional()`, который принимает в качестве первого аргумента 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 по прогнозированию температуры.

### Листинг 10.23 Обучение и оценка двунаправленной LSTM

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))
outputs <- inputs %>%
  bidirectional(layer_lstm(units = 16)) %>% ←
  layer_dense(1)
model <- keras_model(inputs, outputs)

model %>% compile(optimizer = "rmsprop",
                 loss = "mse",
                 metrics = "mae")

history <- model %>%
  fit(train_dataset,
      epochs = 10,
      validation_data = val_dataset)
```

Обратите внимание,  
что `layer_lstm()` не получает  
входы напрямую

Качество этой модели ничуть не улучшилось, по сравнению с обычным слоем `layer_lstm()`. Легко понять, почему: все прогностические способности исходят из половины сети, обрабатывающей данные в прямом хронологическом порядке, потому что, как мы уже выяснили, качество половины, обрабатывающей данные в обратном порядке, сильно отстает в этой задаче (потому что в данном случае недавнее прошлое имеет большее значение, чем отдаленное). В то же время наличие этой половины удваивает емкость сети и приводит к тому, что она гораздо раньше начинает переобучаться.

Однако двунаправленные RNN отлично подходят для текстовых данных или любых других данных, где важно наличие порядка внутри последовательности, но не имеет значения, *какой именно* из двух порядков вы выбрали – прямой или обратный. Фактически какое-то время в 2016 году двунаправленные LSTM считались передовым решением для многих задач обработки естественного языка (до появления архитектуры Transformer, о которой вы узнаете 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 выбрать другой оптимизатор;
- попробовать использовать стек слоев `layer_dense()` в качестве регрессора поверх рекуррентного слоя вместо одного `layer_dense()`;
- улучшить входные данные для модели: попробовать использовать более длинные или короткие последовательности или другую частоту дискретизации либо начать конструировать признаки исходя из априорных знаний о задаче.

Как всегда, глубокое обучение – это больше искусство, чем наука. Мы можем дать рекомендации, подсказав, какие приемы могут улучшить или нет 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.

По моему опыту, лучшее, что вы можете сделать с этим набором данных о погоде, – превзойти точность базовой модели без глубокого обучения примерно на 10 %. Это не очень хороший, но достаточно очевидный результат: погода в ближайшем будущем очень предсказуема, если у вас есть доступ к данным от обширной сети метеостанций из разных мест, но плохо предсказуема, если у вас есть измерения только из одного места. Изменение погоды в том месте, где вы находитесь, зависит от текущих погодных условий в других географических регионах.

### Рынки и машинное обучение

Некоторые читатели наверняка захотят воспользоваться приемами, представленными здесь, для прогнозирования стоимости ценных бумаг на фондовом рынке (обменных курсов валют и т. д.). Рынки имеют совершенно иные статистические характеристики, чем природные явления, такие как погода. Когда дело доходит до рынков, данные о прошлом не являются хорошей основой для прогнозов; невозможно пытаться ехать вперед, глядя в зеркало заднего вида. С другой стороны, машинное обучение оправданно применять к наборам данных, когда прошлое служит хорошим предсказателем будущего, например в таких случаях, как прогноз погоды, потребление электроэнергии или посещаемость магазина.

Нельзя забывать, что любая торговля на фондовых рынках – это, по сути, информационная гонка: получение преимущества за счет использования данных или идей, которых нет у других участников рынка. Попытка использовать известные методы машинного обучения и общедоступ-

ные данные для победы на рынке фактически ведет в тупик, потому что у вас не будет никакого информационного преимущества по сравнению с остальными игроками. Скорее всего, вы потратите впустую свое время и силы, ничего не добившись.

## Краткие итоги главы

- Как было сказано в главе 5, приступая к решению новой задачи, желательно сначала определить базовый уровень для выбранной вами метрики. Если у вас нет обоснованного базового уровня, который должна превзойти модель, вы не можете сказать, добились ли вы успеха.
- Прежде чем приступать к разработке дорогих и сложных моделей, попробуйте самые простые решения, чтобы убедиться, что дополнительные расходы оправданы. Иногда простая модель оказывается лучшим вариантом.
- Если порядок ваших данных имеет значение (в частности, для данных временных рядов), рекуррентные сети легко превосходят модели, которые удаляют составляющую времени. Два основных слоя RNN, доступных в Keras, – это LSTM и GRU.
- Чтобы использовать прореживание с рекуррентными сетями, вы должны применить маску прореживания с постоянным временем и рекуррентную маску прореживания. Они встроены в рекуррентные слои Keras, поэтому все, что вам нужно сделать, – это использовать аргументы `recurrent_dropout` рекуррентных слоев.
- Многослойные RNN обеспечивают большую репрезентативную мощность, чем один слой RNN. Но при этом они намного дороже в вычислительном отношении и, следовательно, не всегда того стоят. Хотя они дают явные преимущества при решении сложных задач (таких как машинный перевод), их не всегда целесообразно применять к более мелким и простым задачам.

# 11

## Глубокое обучение в обработке текстов

---

**Эта глава охватывает следующие темы:**

- предварительная обработка текста для приложений машинного обучения;
- различные методы обработки текста;
- архитектура Transformer;
- преобразование последовательностей.

### 11.1 **Обработка естественного языка: обзор отрасли**

В компьютерных науках принято называть человеческие языки, такие как английский или китайский, *естественными языками*, чтобы отличать их от «искусственных» языков, разработанных для машин, таких как ассемблер, LISP или XML. У каждого машинного языка есть один или несколько создателей: обычно это разработчик набора формальных правил, описывающих, какие операторы вы можете использовать в этом языке и что они означают. Сначала появляются правила, а люди начинают использовать машинный язык только после того, как набор правил завершен и проверен. С естественным человеческим языком все наоборот: сначала идет использование, а потом появляются правила. Естественный язык сформировался

в процессе эволюции, как и биологические организмы, – вот что делает его «естественным». Его «правила», как и грамматика того или иного языка, были формализованы постфактум и часто игнорируются или нарушаются пользователями. В результате машиночитаемый язык является высокоструктурированным и строгим; он использует строгие синтаксические правила для объединения точно определенных понятий из фиксированного словаря. Естественный язык всегда остается беспорядочным – двусмысленным, хаотичным, растянутым и постоянно изменяющимся.

Создание алгоритмов, способных понимать естественный язык, – грандиозная задача, ведь язык (и в особенности текст) лежит в основе большинства наших коммуникаций и нашей культурной среды. Интернет преимущественно текстовый. Язык – это основной способ хранить наши знания. Сами наши мысли в значительной степени основаны на языке. Однако способность понимать естественный язык долгое время ускользала от машин. Когда-то исследователи наивно полагали, что можно просто формализовать «набор правил языка», подобно тому, как можно записать набор правил LISP. Поэтому ранние попытки построить системы *обработки естественного языка* (natural language processing, NLP) предпринимались через призму «прикладной лингвистики». Инженеры и лингвисты вручную создавали сложные наборы правил для выполнения базового машинного перевода или создания простых чат-ботов, таких как знаменитая программа ELIZA 1960-х годов, которая использовала сопоставление с образцом для поддержания простейшего разговора. Но язык – капризная штука, он плохо поддается формализации. После нескольких десятилетий непрерывных усилий возможности этих систем так и остались разочаровывающими.

Подходы на основе созданных вручную правил доминировали вплоть до 1990-х годов. Но начиная с конца 1980-х более быстрые компьютеры и большие объемы доступных данных открыли другой путь к решению задачи. Осознав, что лингвистические системы представляют собой нагромождение специальных правил, разработчики начали задавать вопросы: «Можно ли использовать корпус данных для автоматизации процесса поиска этих правил? Можно ли искать правила в каком-то пространстве правил, вместо того чтобы придумывать их самому?» Отсюда остается буквально один шаг до машинного обучения, и неудивительно, что в конце 1980-х появились первые ростки этой технологии в обработке естественного языка. Самые ранние из них были основаны на деревьях решений – подход заключался в прямой автоматизации выработки правил «если/то/иначе» предыдущих систем. Затем стали набирать обороты статистические подходы, начиная с логистической регрессии. Со временем обучаемые параметрические модели полностью взяли верх, и лингвистика стала рассматриваться скорее как помеха, чем как полезный инструмент. Фредерик Елинек, один из первых исследователей распознавания речи, в 1990-х пошутил: «Каж-

дый раз, когда я увольняю лингвиста, точность распознавателя речи повышается».

Именно в этом суть современного NLP: использование машинного обучения и больших наборов данных не для того, чтобы добиться от компьютеров *понимания* естественного языка на уровне человека (что и сейчас является почти недостижимой целью), а для того, чтобы дать компьютерам возможность принимать фрагмент языка в качестве входных данных и возвращать что-то полезное, например предсказанные ответы на следующие вопросы:

- «Какова тема этого текста?» (классификация текстов);
- «Этот текст содержит оскорбления?» (фильтрация контента);
- «Этот текст звучит одобрительно или осуждающе?» (анализ эмоциональной окраски);
- «Каким должно быть следующее слово в этом незаконченном предложении?» (моделирование языка);
- «Как это сказать по-немецки?» (перевод);
- «Как изложить смысл этой статьи в одном абзаце?» (резюмирование);
- ... и т. д.

Конечно, нужно помнить, что ни одна из этих моделей в действительности не понимает текст в человеческом смысле; они лишь отражают статистическую структуру письменного языка – этого достаточно для решения многих простых задач обработки текста. Глубокое обучение для обработки естественного языка – это распознавание образов на уровне слов, предложений и абзацев, примерно так же, как компьютерное зрение – это распознавание образов на уровне пикселей и областей изображения.

Набор инструментов NLP – в первую очередь деревья решений и логистическая регрессия – медленно эволюционировал с 1990-х по начало 2010-х годов. Исследования были в основном сосредоточены на конструировании признаков. Когда я выиграл свое первое соревнование по NLP на Kaggle в 2013 году, моя модель, как вы уже догадались, была основана на деревьях решений и логистической регрессии. Однако примерно в 2014–2015 годах настало время перемен. Исследователи начали применять для обработки естественного языка рекуррентные нейросети, в частности LSTM – алгоритм обработки последовательностей, появившийся в конце 1990-х годов и долгое время остававшийся незамеченным.

В начале 2015 года платформа Keras предоставила первую простую в использовании реализацию LSTM с открытым исходным кодом – как раз в начале всплеска интереса к рекуррентным нейронным сетям. До этого существовали только экспериментальные решения, которые сложно было использовать повторно. Далее, с 2015 по 2017 год, в бурно развивающейся области NLP начали доминировать рекуррентные нейронные сети. Двухнаправленные модели LSTM, в частности, определяют современный уровень во многих важных задачах, от обобщения до ответов на вопросы и машинного перевода.

Наконец, примерно в 2017–2018 годах на смену RNN пришла новая архитектура Transformer, о которой вы узнаете во второй половине этой главы. Сети-трансформеры позволили добиться значительного прогресса за короткий период времени, и сегодня большинство систем NLP основаны на архитектуре Transformer.

Теперь мы можем перейти к подробностям. В этой главе мы рассмотрим все этапы развития NLP – от самых основ до машинного перевода с помощью Transformer.

## 11.2 Подготовка текстовых данных

Модели глубокого обучения представляют собой дифференцируемые функции, поэтому могут обрабатывать только числовые тензоры: они не способны принимать необработанный текст в качестве входных данных. *Векторизация текста* – это процесс преобразования текста в числовые тензоры. Процессы векторизации текста бывают реализованы по-разному, но все они работают по одной и той же схеме (рис. 11.1):

- *стандартизация* текста для упрощения дальнейшей обработки, например перевод в нижний регистр или удаление знаков препинания;

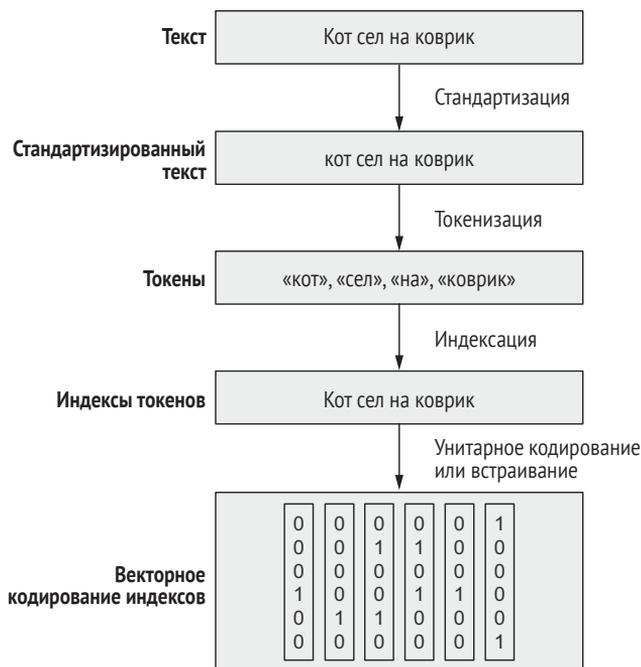


Рис. 11.1 Этапы векторизации исходного текста

- разбиение текста на блоки (называемые токенами), такие как символы, слова или группы слов. Этот шаг также называют *токенизацией*;
- *конвертация* каждого такого токена в числовой вектор. Обычно это подразумевает предварительную *индексацию* всех токенов, присутствующих в данных.

Давайте рассмотрим подробнее каждый из этих шагов.

### 11.2.1 Стандартизация текста

Рассмотрим два предложения на английском языке:

- «sunset came. i was staring at the Mexico sky. Isnt nature splendid??» (*наступил закат. я смотрел на небо Мексики. Разве природа не прекрасна??*);
- «Sunset came; I stared at the México sky. Isn't nature splendid?» (*Наступил закат; Я смотрел на мексиканское небо. Разве природа не прекрасна?*).

Эти предложения очень похожи – на самом деле они почти идентичны. Тем не менее если вы преобразуете их в байтовые строки, то получатся очень разные представления, потому что «i» и «I» – это два разных символа, «Mexico» и «México» – два разных слова, «isnt» и «isn't» – не одно и то же, и т. д. Модель машинного обучения априори не знает, что «i» и «I» – это одна и та же буква, что «é» – это «е» с ударением или что «staring» и «stared» – это две формы одного глагола.

Стандартизация текста – это базовая форма конструирования признаков, целью которой является устранение различий в кодировании, которые не должны поступить в модель. Это касается не только машинного обучения – вам пришлось бы делать то же самое, если бы вы создавали поисковую систему.

Одна из самых простых и распространенных схем стандартизации – перевести все символы в нижний регистр и удалить знаки препинания. Два исходных предложения после стандартизации будут выглядеть так:

- sunset came i was staring at the mexico sky isnt nature splendid;
- sunset came i stared at the méxico sky isnt nature splendid.

Уже гораздо лучше. Другим распространенным преобразованием является преобразование специальных символов в стандартную форму, например замена «é» на «е», «æ» на «ae» и т. д. Тогда наш токен «méxico» превратится в «mexico».

Наконец, существует гораздо более продвинутый способ стандартизации, который реже используется в контексте машинного обучения, – *стемминг* (stemming). Это преобразование вариантов слова (например, различных форм глагола) в единое общее представление, например приведение слов «ловил» и «поймал» к общей

начальной форме «ловить». После стемминга «was staring» и «stared» станут чем-то вроде «[stare]», и наши два исходных предложения наконец приобретут одинаковый вид:

- sunset came i [stare] at the mexico sky isnt nature splendid.

Благодаря этим методам стандартизации вашей модели потребуются меньше данных для обучения, и она будет лучше обобщать – ей не понадобятся многочисленные примеры различных форм слов, чтобы понять, что они означают одно и то же, и она сможет извлечь смысл из слова «México», даже если встречала в своем обучающем наборе только слово «mexico». Конечно, стандартизация может также удалить часть информации, поэтому всегда помните о контексте: например, если вы пишете модель, извлекающую вопросы из статей с интервью, она обязательно должна обрабатывать знак вопроса «?» как отдельный токен, а не отбрасывать его, потому что это полезный сигнал для данной конкретной задачи.

## 11.2.2 Разделение текста (токенизация)

После стандартизации текста его необходимо разбить на минимальные элементы для векторизации (токены). Этот этап называется *токенизацией*. У вас есть на выбор три способа:

- *токенизация на уровне слов* – токены представляют собой подстроки, разделенные пробелами (или знаками препинания). Возможно также дальнейшее разбиение слов на подслова, когда это допустимо, например обработка «подбежать» как «под + бежать»;
- *токенизация N-грамм* – когда токены представляют собой группы из  $N$  последовательных слов. Например, «этот кот» или «он был» будут токенами в виде 2-грамм (их также называют биграммami);
- *побуквенная токенизация* – каждая буква является отдельным токеном. На практике эта схема используется редко, и вы можете встретить ее только в специализированных применениях, таких как генерация текста или распознавание речи.

Как правило, вы будете использовать токенизацию на уровне слов или  $N$ -грамм. Существует два вида моделей обработки текста: *модель последовательности* сохраняет и учитывает порядок слов, а *модель мешка слов* обрабатывает множество слов, отбрасывая их первоначальный порядок. Если вы создаете модель последовательности, вы будете использовать токенизацию на уровне слов, а если вы создаете модель мешка слов, вы будете использовать токенизацию на уровне  $N$ -грамм. *N-граммы* – это способ искусственно ввести в модель небольшое количество информации о порядке слов. В этой главе вы познакомитесь поближе с каждой разновидностью моделей и узнаете, в каких случаях их следует применять.

### Что такое $N$ -граммы и мешки слов

$N$ -граммы – это группы из  $N$  (или менее) последовательных слов, которые можно извлечь из предложения. Та же идея применима к символам вместо слов.

Вот простой пример. Рассмотрим предложение «The cat sat on the mat» («Кошка села на коврик»). Его можно разложить на следующий набор 2-грамм (или биграмм):

```
c("the", "the cat", "cat", "cat sat", "sat",
  "sat on", "on", "on the", "the mat", "mat")
```

Также его можно разложить на такой набор 3-грамм (или триграмм):

```
c("the", "the cat", "cat", "cat sat", "the cat sat",
  "sat", "sat on", "on", "cat sat on", "on the",
  "sat on the", "the mat", "mat", "on the mat")
```

Такие наборы называют *мешком биграмм* или *мешком триграмм* соответственно. Термин мешок в данном случае отражает тот факт, что вы имеете дело с множеством токенов, а не со списком или последовательностью: токены в мешке не упорядочены. Это семейство методов токенизации называют *мешком слов* (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 общую структуру предложений), этот метод обычно используется в поверхностных моделях обработки естественного языка и крайне редко – в моделях глубокого обучения. Извлечение  $N$ -грамм – это еще одна форма конструирования признаков, но в моделях глубокого обучения этот неудобный метод заменяют на конструирование иерархических признаков. Одномерные сверточные и рекуррентные нейронные сети и трансформеры способны получать представления для групп слов и символов без явного определения таких групп, просматривая последовательности слов или символов.

### 11.2.3 Индексация словаря

После разбиения текста на токены нужно закодировать каждый токен в числовое представление. Потенциально вы могли бы сделать это без сохранения состояния, например путем хеширования каждой лексемы в фиксированный двоичный вектор, но на практике процесс обычно выглядит иначе – нужно построить индекс всех терминов, найденных в обучающих данных (так называемый *словарь*), и назначить уникальное целое число каждой записи в словаре, например так:

```
vocabulary <- character()
for (string in text_dataset) {
  tokens <- string %>%
```

```

    standardize() %>%
    tokenize()
  vocabulary <- unique(c(vocabulary, tokens))
}

```

Затем вы можете преобразовать позицию целочисленного индекса в векторную кодировку, которая может быть обработана нейронной сетью, например в прямой унитарный код:

```

one_hot_encode_token <- function(token) {
  vector <- array(0, dim = length(vocabulary))
  token_index <- match(token, vocabulary)
  vector[token_index] <- 1
  vector
}

```

Нужно заметить, что на этом этапе обычно оставляют словарь только из 20 000 или 30 000 самых распространенных слов, найденных в обучающих данных. Любой набор текстовых данных обычно содержит чрезвычайно большое количество уникальных терминов, большинство из которых встречается только один или два раза. Индексация этих редких терминов привела бы к построению чрезмерно обширного пространства признаков, большинство из которых не содержат полезной информации.

Помните, как вы обучали свои первые модели на наборе данных IMDB в главах 4 и 5? Данные, которые вы взяли из `dataset_imdb()`, уже были предварительно переработаны в последовательности целых чисел, где каждое целое число означало определенное слово. Тогда мы использовали параметр `num_words = 10000`, чтобы ограничить словарь первыми десятью тысячами самых распространенных слов, найденных в обучающих данных.

Здесь есть важная деталь, которую нельзя упускать из виду: когда мы ищем новый токен в указателе нашего словаря, он не обязательно там найдется. В ваших обучающих данных может отсутствовать слово «черимойя» (или, возможно, вы исключили его из своего индекса, потому что оно было слишком редким), поэтому выполнение `token_index = match("черимойя", vocabulary)` может вернуть NA. Чтобы справиться с этим, вы должны использовать индекс «вне словаря» (*индекс OOV* – out of vocabulary) – универсальный для любого токена, которого не было в указателе. Обычно это индекс 1, и на самом деле вы будете использовать команду `token_index = match("черимойя", word, nomatch = 1)`. При декодировании последовательности целых чисел обратно в слова вы замените 1 на что-то вроде «[UNK]».

Вы можете спросить, почему в этом случае используют 1, а не 0. Дело в том, что индекс 0 уже занят. Обычно используются два специальных токена: токен OOV (индекс 1) и *токен маски* (индекс 0). Токен OOV означает «здесь было слово, которое мы не узнали», а токен маски говорит: «игнорируйте меня, я не слово». Вы можете использовать токен маски, в частности для дополнения последова-



```

    self
  }

vectorizer <- new_vectorizer()
dataset <- c("I write, erase, rewrite", ← Хайку поэта Хокусэи
            "Erase again, and then",
            "A poppy blooms.")
vectorizer$make_vocabulary(dataset)

```

Так выглядит результат работы этого кода:

```

test_sentence <- "I write, rewrite, and still rewrite again"
encoded_sentence <- vectorizer$encode(test_sentence)
print(encoded_sentence)

```

```
[1] 2 3 5 7 1 5 6
```

```

decoded_sentence <- vectorizer$decode(encoded_sentence)
print(decoded_sentence)

```

```
[1] "i" "write" "rewrite" "and" "[UNK]" "rewrite" "again"
```

Однако этот код не очень эффективен. На практике вы будете работать с более эффективным слоем Keras `layer_text_vectorization()`, который можно добавить непосредственно в конвейер набора данных TensorFlow или модель Keras. Использование `layer_text_vectorization()` выглядит так:

```

text_vectorization <-
  layer_text_vectorization(output_mode = "int") ←

```

Настройка слоя для возврата последовательностей слов,  
закодированных как целые индексы. Доступно несколько  
других режимов вывода, которые вы вскоре увидите в действии

По умолчанию `layer_text_vectorization()` будет использовать настройки «преобразовать в нижний регистр и удалить знаки препинания» для стандартизации текста и «разделить по пробелам» для токенизации. Но что особенно важно, вы можете предоставить пользовательские функции для стандартизации и токенизации, а это означает, что слою хватит гибкости, чтобы справиться с любым вариантом использования. Учтите, что такие пользовательские функции должны работать с тензорами dtype `tf.string`, а не с обычными векторами символов R! Например, поведение слоя по умолчанию эквивалентно следующему коду:

```

library(tensorflow)
custom_standardization_fn <- function(string_tensor) {
  string_tensor %>%
    tf$string$lower() %>% ← Перевод строк в нижний регистр
    tf$string$regex_replace("[[:punct:]]", "") ← Замена символов
}

```

пунктуации пустой строкой

```

custom_split_fn <- function(string_tensor) {
  tf$string$split(string_tensor) ← Разделение строк по пробелам
}

text_vectorization <- layer_text_vectorization(
  output_mode = "int",
  standardize = custom_standardization_fn,
  split = custom_split_fn
)

```

Чтобы проиндексировать словарь текстового корпуса, просто вызовите метод `adapt()` слоя с объектом набора данных TensorFlow, который возвращает строки, или с вектором символов R:

```

dataset <- c("I write, erase, rewrite",
            "Erase again, and then",
            "A poppy blooms.")
adapt(text_vectorization, dataset)

```

Для получения готового словаря применяется метод `get_vocabulary()`. Он может быть полезен, если вам нужно преобразовать текст, закодированный как целочисленная последовательность, обратно в слова. Первые две записи в словаре – это токен маски (индекс 0) и токен OOV (индекс 1). Записи в словарном списке отсортированы по частоте, поэтому в реальном наборе данных самые распространенные слова английского языка, такие как «the» или «a», будут стоять первыми.

### Листинг 11.1 Отображение словаря

```

get_vocabulary(text_vectorization)

[1] "" "[UNK]" "erase" "write" "then" "rewrite" "poppy"
[8] "i" "blooms" "and" "again" "a"

```

В качестве примера попробуем закодировать и раскодировать предложение:

```

vocabulary <- text_vectorization %>% get_vocabulary()
test_sentence <- "I write, rewrite, and still rewrite again"
encoded_sentence <- text_vectorization(test_sentence)
decoded_sentence <- paste(vocabulary[as.integer(encoded_sentence) + 1],
                        collapse = " ")

```

```
encoded_sentence
```

```
tf.Tensor([ 7 3 5 9 1 5 10], shape=(7), dtype=int64)
```

```
decoded_sentence
```

```
[1] "i write rewrite and [UNK] rewrite again"
```

## Использование `layer_text_vectorization()` в конвейере набора данных TensorFlow или как компонента модели

Поскольку `layer_text_vectorization()` – это в основном операция поиска в словаре, которая преобразует токены в целые числа, ее нельзя выполнять на GPU (или TPU) – только на CPU. Если вы обучаете свою модель на графическом процессоре, необходимо выполнить `layer_text_vectorization()` на обычном процессоре, прежде чем отправлять свои выходные данные на графический процессор. Это ограничение может существенно повлиять на производительность.

Есть два способа использования нашего `layer_text_vectorization()`. Первый вариант – поместить его в конвейер набора данных TensorFlow, например:

```
int_sequence_dataset <- string_dataset %>%
  dataset_map(text_vectorization,
              num_parallel_calls = 4)
```

`string_dataset` будет набором данных TF, который дает строковые тензоры

Аргумент `num_parallel_calls` используется для распараллеливания вызова `dataset_map()` между несколькими ядрами ЦП

Второй вариант – сделать этот слой частью модели (ведь это фактически слой Keras), как показано в следующем фрагменте псевдокода:

```
text_input <- layer_input(shape = shape(), dtype = "string")
vectorized_text <- text_vectorization(text_input)
embedded_input <- vectorized_text %>% layer_embedding(...)
output <- embedded_input %>% ...
model <- keras_model(text_input, output)
```

Символьный ввод, ожидающий строку

Применяем к вводу слой векторизации текста

Вы можете продолжить размещать сверху новые слои – это обычная модель Functional API

Между этими вариантами есть важное различие: если этап векторизации является частью модели, он будет выполняться синхронно с остальной частью модели. Это означает, что на каждом этапе обучения остальная часть модели (выполняемая на GPU) должна будет ждать, пока выходные данные слоя `layer_text_vectorization()` (выполняемого на CPU) будут готовы, прежде чем она сможет продолжить работу. С другой стороны, размещение слоя в конвейере набора данных TensorFlow позволяет вам выполнять асинхронную предварительную обработку ваших данных: пока GPU выполняет модель на готовом пакете векторизованных данных, CPU занимается векторизацией следующего пакета необработанных строк.

Если вы обучаете модель на GPU или TPU, то наверняка предпочтете первый вариант, чтобы получить максимальную производительность. Мы будем применять этот подход во всех практических примерах в данной главе. Однако при обучении на CPU можно использовать синхронную об-

работку: вы получите 100%-ное использование ядер процессора независимо от того, какой вариант выберете.

При экспорте готовой модели в производственную среду вы должны отправить модель, которая принимает в качестве входных данных необработанные строки, как в фрагменте кода для второго варианта выше; в противном случае вам пришлось бы отдельно реализовывать стандартизацию и токенизацию текста в вашей производственной среде (может быть, в JavaScript?) и вы столкнулись бы с риском появления небольших расхождений в предварительной обработке, которые снижают точность модели. К счастью, `layer_text_vectorization()` позволяет вам включить предварительную обработку текста прямо в вашу модель, упрощая ее развертывание, даже если вы изначально использовали слой как часть конвейера набора данных TensorFlow. Позже вы узнаете, как экспортировать обученную модель для логического вывода, включающую предварительную обработку текста.

На этом мы заканчиваем обсуждение предварительной обработки текста и переходим к этапу моделирования.

### 11.3 Два подхода к представлению групп слов: наборы и последовательности

Вопрос о том, как модель машинного обучения должна представлять *отдельные слова*, не вызывает особых споров: это категориальные признаки (значения из предопределенного множества), и мы знаем, как с ними обращаться. Они должны быть закодированы как измерения в пространстве признаков или как векторы категорий (в данном случае векторы слов). Однако гораздо более проблематичный вопрос заключается в том, как закодировать *способ встраивания слов в предложения*: порядок слов.

Проблема порядка слов в естественном языке уникальна – в отличие от шагов временного ряда, слова в предложении не имеют однозначного, канонического порядка. В разных языках похожие слова упорядочиваются по-разному. Например, структура предложений в английском языке сильно отличается от японского. Даже в одном языке вы обычно можете сказать одно и то же по-разному, немного переставив слова. Более того, если вы полностью хаотично перемешаете слова в коротком предложении, вы все равно сможете понять его смысл, хотя во многих случаях может возникнуть значительная двусмысленность. Порядок слов, несомненно, важен, но его связь со значением предложения неоднозначна.

Способ представления порядка слов – ключевой вопрос, из которого вытекают различные виды архитектур NLP. Самое простое, что вы можете сделать, – игнорировать порядок и рассматривать текст

как неупорядоченный набор слов (модели мешка слов). Вы также можете решить, что слова должны обрабатываться строго в том порядке, в котором они появляются, по одному, как шаги во временном ряду, – тогда вы можете использовать рекуррентные модели из предыдущей главы. Наконец, возможен и гибридный подход: архитектура Transformer технически независима от порядка, но она вводит информацию о позиции слова в обрабатываемые представления, что позволяет ей одновременно рассматривать разные части предложения (в отличие от RNN), сохраняя при этом осведомленность о порядке. Следовательно, RNN и Transformer относятся к моделям последовательностей.

Исторически сложилось так, что самые ранние реализации машинного обучения в области NLP основывались только на моделях мешка слов. Интерес к моделям последовательностей возник в 2015 году, с возрождением рекуррентных нейронных сетей. Сегодня актуальны оба подхода. Далее мы рассмотрим, как они работают и когда их лучше использовать.

Мы продемонстрируем каждый подход на известном примере классификации текста: наборе данных классификации эмоциональной окраски IMDB. В главах 4 и 5 вы работали с предварительно векторизованной версией набора данных IMDB; теперь давайте возьмем необработанные текстовые данные IMDB точно так же, как если бы вы взялись за решение новой задачи классификации текста в реальном мире.

### 11.3.1 Подготовка данных обзоров фильмов IMDB

Давайте начнем с загрузки и распаковки набора данных со страницы Эндрю Мааса в Стэнфорде:

```
url <- "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
filename <- basename(url)
options(timeout = 60 * 10) ← 10-минутный период ожидания
download.file(url, destfile = filename)
untar(filename)
```

У вас должен получиться каталог с именем `aclImdb` со следующей структурой:

```
fs::dir_tree("aclImdb", recurse = 1, type = "directory")
```

```
aclImdb
├── test
│   ├── neg
│   └── pos
└── train
    ├── neg
    └── pos
```

Например, каталог `train/pos/` содержит набор из 12 500 текстовых файлов с положительными отзывами о фильмах, которые будут использоваться в качестве обучающих данных. Отрицательные отзывы размещены в каталоге `train/neg/`. Всего имеется 25 000 текстовых файлов для обучения и еще 25 000 для тестирования.

Там также есть подкаталог `train/unsup`, который нам не нужен. Просто удалите его:

```
fs::dir_delete("aclImdb/train/unsup/")
```

Теперь стоит взглянуть на содержимое нескольких текстовых файлов. Независимо от того, работаете ли вы с текстом или изображениями, не забывайте проверять, как выглядят ваши данные, прежде чем погрузиться в их моделирование. Это закрепит ваше понимание того, как на самом деле работает ваша модель:

```
writelines(readLines("aclImdb/train/pos/4077_10.txt", warn = FALSE))
```

```
I first saw this back in the early 90s on UK TV, i did like it then but i missed the chance to tape it, many years passed but the film always stuck with me and i lost hope of seeing it TV again, the main thing that stuck with me was the end, the hole castle part really touched me, its easy to watch, has a great story, great music, the list goes on and on, its OK me saying how good it is but everyone will take there own best bits away with them once they have seen it, yes the animation is top notch and beautiful to watch, it does show its age in a very few parts but that has now become part of it beauty, i am so glad it has came out on DVD as it is one of my top 10 films of all time. Buy it or rent it just see it, best viewing is at night alone with drink and food in reach so you don't have to stop the film.<br /><br />Enjoy
```

(Я впервые увидел фильм еще в начале 90-х по британскому телевидению, тогда он мне понравился, но я упустил шанс записать его на пленку, прошло много лет, но фильм навсегда остался со мной, и я потерял надежду снова увидеть его по телевидению, больше всего мне запомнился финал, эпизод со старым замком глубоко тронул меня, фильм легко смотреть, у него отличный сюжет, отличная музыка, список можно продолжать и продолжать, я искренне говорю, насколько он хорош, каждый найдет в нем для себя что-то лучшее, да, съемка на высшем уровне и это просто красиво, хотя местами чувствуется возраст фильма, но теперь это стало частью его очарования, я так рад, что фильм вышел на DVD, так как для меня он входит в десятку лучших фильмов всех времен. Купите его или возьмите напрокат, просто посмотрите, лучше всего ночью в одиночестве, когда напитки и еда под рукой, чтобы не пришлось останавливать фильм.<br /><br />Наслаждайтесь)

Теперь подготовим проверочный набор, разместив 20 % обучающих текстовых файлов в новом каталоге `aclImdb/val`. Как и прежде, мы будем использовать пакет `fs` R:

```

library(fs)
set.seed(1337)
base_dir <- path("aclImdb")
for (category in c("neg", "pos")) {
  filepaths <- dir_ls(base_dir / "train" / category)
  num_val_samples <- round(0.2 * length(filepaths))
  val_files <- sample(filepaths, num_val_samples)
  dir_create(base_dir / "val" / category)
  file_move(val_files,
            base_dir / "val" / category)
}

```

Задаем начальное значение рандомизатора, чтобы получать один и тот же проверочный набор из вызова sample() каждый раз, когда мы запускаем код

Выделяем 20 % обучающих данных в проверочный набор

Переносим файлы в aclImdb/val/neg и aclImdb/val/pos

Помните, как в главе 8 мы использовали утилиту image\_dataset\_from\_directory() для создания пакетного набора данных TensorFlow из изображений и их меток с определенной структурой каталогов? Вы можете сделать то же самое для текстовых файлов, используя утилиту text\_dataset\_from\_directory(). Создадим три объекта набора данных – для обучения, проверки и контроля – при помощи следующего кода:

```

library(keras)
library(tfdatasets)
train_ds <- text_dataset_from_directory("aclImdb/train")
val_ds <- text_dataset_from_directory("aclImdb/val")
test_ds <- text_dataset_from_directory("aclImdb/test")

```

Эта строка должна вывести «Найдено 20 000 файлов, принадлежащих 2 классам»; если же вы видите «Найдено 70 000 файлов, принадлежащих 3 классам», это означает, что вы забыли удалить каталог aclImdb/train/unsup

Размер batch\_size по умолчанию равен 32. Если вы сталкиваетесь с ошибками нехватки памяти при обучении моделей на своем компьютере, попробуйте меньший размер пакета: text\_dataset\_from\_directory(«aclImdb/train», batch\_size = 8)

В эти наборы будут помещены входные данные, которые являются тензорами TensorFlow tf.string, и цели, которые представляют собой тензоры int32, кодирующие значение «0» или «1».

**Листинг 11.2 Отображение форм и типов первого пакета**

```

c(inputs, targets) %<-% iter_next(as_iterator(train_ds))
str(inputs)

| <tf.Tensor: shape=(32), dtype=string, numpy=...>

str(targets)

| <tf.Tensor: shape=(32), dtype=int32, numpy=...>

inputs[1]

| tf.Tensor(b'Let me start by saying that I\'d read a number of reviews before
renting this film and kind of knew what to expect. Still, I was surprised by
just how bad it was. <br /><br />I am a big werewolf fan, and have grown

```

```

...
Otherwise, give this one a miss.', shape=(), dtype=string)

targets[1]

tf.Tensor(0, shape=(), dtype=int32)

```

На этом подготовка завершена. Далее мы займемся обучением моделей на этих данных.

### 11.3.2 Обработка слов без учета порядка

Самый простой способ закодировать фрагмент текста для обработки моделью машинного обучения – отбросить информацию о порядке и рассматривать его как набор («мешок») токенов. Мы можем либо рассматривать отдельные слова (униграммы), либо попытаться восстановить некоторую информацию о локальном порядке, рассматривая группы последовательных токенов ( $N$ -граммы).

#### ОТДЕЛЬНЫЕ СЛОВА (УНИГРАММЫ) С БИНАРНЫМ КОДИРОВАНИЕМ

Если вы используете набор отдельных слов, предложение «the cat sat on the mat» становится вектором символов, где мы игнорируем порядок:

```
c("cat", "mat", "on", "sat", "the")
```

Основное преимущество этой кодировки заключается в том, что вы можете представить весь текст в виде одного вектора, каждый элемент которого является индикатором присутствия для определенного слова. Например, используя *множественное унитарное кодирование* (multi-hot encode), можно закодировать текст в виде вектора с количеством измерений, равным количеству слов в вашем словаре, с нулями почти везде и несколькими единицами в позициях, которые соответствуют словам, присутствующим в тексте. Мы уже применяли такой подход, когда работали с текстовыми данными в главах 4 и 5, а теперь перенесем его на новую задачу.

Сначала обработаем наши наборы исходных текстовых данных с помощью слоя `layer_text_vectorization()`, чтобы они давали векторы двоичных слов со множественным унитарным кодированием. Наш слой будет рассматривать только отдельные слова (то есть *униграммы*).

#### Листинг 11.3 Предварительная обработка наборов данных с помощью `layer_text_vectorization()`

```

text_vectorization <-
  layer_text_vectorization(max_tokens = 20000,

```

Ограничим словарь 20 000 наиболее часто встречающихся слов. В противном случае мы бы индексировали каждое слово в обучающих данных – потенциально десятки тысяч терминов, которые встречаются только один или два раза и, следовательно, не являются информативными. В большинстве случаев 20 000 – это подходящий размер словарного запаса для классификации текстов

```

Закодируем выходные токены
как множественные унитарные векторы
output_mode = "multi_hot")

text_only_train_ds <- train_ds %>%
  dataset_map(function(x, y) x)
  ← Подготовим набор данных, который
  содержит только необработанные
  текстовые входные данные (без меток)

adapt(text_vectorization, text_only_train_ds)
  ← Используем этот
  набор данных для
  индексации словаря
  набора данных
  с помощью метода
  adapt()

binary_1gram_train_ds <- train_ds %>%
  dataset_map( ~ list(text_vectorization(.x), .y),
               num_parallel_calls = 4)
binary_1gram_val_ds <- val_ds %>%
  dataset_map( ~ list(text_vectorization(.x), .y),
               num_parallel_calls = 4)
binary_1gram_test_ds <- test_ds %>%
  dataset_map( ~ list(text_vectorization(.x), .y),
               num_parallel_calls = 4)

Подготовка обработанных версий обучающих, проверочных и контрольных данных.
Обязательно укажите параметр num_parallel_calls, чтобы использовать несколько ядер CPU
    
```

### Определение формулы как функции (~)

В качестве аргумента `map_func` для `dataset_map()` мы передали формулу, определенную с помощью оператора `~`, а не функцию. Если аргумент `map_func` является формулой, например `~ .x + 2`, он будет преобразован в функцию. Существует три способа обращения к аргументам такой функции:

- для функции с одним аргументом используйте `.x`;
- для функции с двумя аргументами используйте `.x` и `.y`;
- чтобы получить дополнительные аргументы, используйте `..1`, `..2`, `..3` и т. д.

Этот синтаксис позволяет создавать очень компактные анонимные функции. Дополнительные сведения и примеры доступны на странице справки R `?purrr::map()`.

Можно посмотреть, как выглядит один из готовых наборов данных (листинг 11.4).

#### Листинг 11.4 Проверка вывода нашего набора данных двоичной униграммы

```

c(inputs, targets) %<-% iter_next(as_iterator(binary_1gram_train_ds))
str(inputs)

| <tf.Tensor: shape=(32, 20000), dtype=float32, numpy=...>

str(targets)

| <tf.Tensor: shape=(32), dtype=int32, numpy=...>
    
```

Входные данные представляют собой пакеты 20 000-мерных векторов. Эти векторы полностью состоят из единиц и нулей

```
inputs[1, ] ←
```

```
tf.Tensor([1. 1. 1. ... 0. 0. 0.], shape=(20000), dtype=float32)
```

```
targets[1]
```

```
tf.Tensor(1, shape=(), dtype=int32)
```

Теперь нам нужно разработать повторно используемую функцию построения модели, которую мы будем использовать во всех наших экспериментах в этом разделе (листинг 11.5).

### Листинг 11.5 Утилита для построения модели

```
get_model <- function(max_tokens = 20000, hidden_dim = 16) {
  inputs <- layer_input(shape = c(max_tokens))
  outputs <- inputs %>%
    layer_dense(hidden_dim, activation = "relu") %>%
    layer_dropout(0.5) %>%
    layer_dense(1, activation = "sigmoid")
  model <- keras_model(inputs, outputs)
  model %>% compile(optimizer = "rmsprop",
                   loss = "binary_crossentropy",
                   metrics = "accuracy")
  model
}
```

Наконец, обучим и протестируем нашу модель.

### Листинг 11.6 Обучение и тестирование модели бинарной униграммы

```
model <- get_model()
model
```

```
Model: "model"
```

```
Layer (type) Output Shape Param #
```

```
=====  
input_1 (InputLayer) [(None, 20000)] 0  
dense_1 (Dense) (None, 16) 320016  
dropout (Dropout) (None, 16) 0  
dense (Dense) (None, 1) 17  
=====
```

```
Total params: 320,033  
Trainable params: 320,033  
Non-trainable params: 0
```

```
callbacks <- list(
  callback_model_checkpoint("binary_1gram.keras", save_best_only = TRUE)
)
```

Мы вызываем `dataset_cache()` для наборов данных, чтобы кешировать их в памяти: таким образом, мы выполним предварительную обработку только один раз, в течение первой эпохи, и будем повторно использовать предварительно обработанные тексты для следующих эпох. Это можно сделать только в том случае, если данные достаточно малы, чтобы поместиться в памяти

```
model %>% fit(
  dataset_cache(binary_1gram_train_ds),
  validation_data = dataset_cache(binary_1gram_val_ds), ←
  epochs = 10,
  callbacks = callbacks
)

model <- load_model_tf("binary_1gram.keras")
cat(sprintf(
  "Test acc: %.3f\n", evaluate(model, binary_1gram_test_ds)["accuracy"]))

| Test acc: 0.887
```

Эта модель демонстрирует точность на контрольных данных 88,7 % – неплохой результат! Нужно учитывать, что поскольку этот конкретный набор данных представляет собой сбалансированную выборку двух классов (положительных отзывов столько же, сколько отрицательных), базовый уровень, которого мы могли бы достичь без обучения реальной модели, составил бы только 50 %. Между тем наилучшая оценка, которую можно получить с этим набором без использования внешних данных, составляет около 95 % на контрольном наборе.

## БИГРАММЫ С ДВОИЧНЫМ КОДИРОВАНИЕМ

Конечно, отказ от порядка слов значительно снижает качество модели, потому что даже атомарные понятия могут быть выражены несколькими словами. Например, термин «Соединенные Штаты» передает понятие, совершенно отличное от значений слов «соединение» и «штат», взятых по отдельности. По этой причине, как правило, в конечном итоге приходится повторно вводить информацию о локальном порядке в представление набора слов, просматривая  $N$ -граммы, а не отдельные слова (чаще всего биграммы).

Представление нашего предложения в виде биграмм выглядит так:

```
c("the", "the cat", "cat", "cat sat", "sat",
  "sat on", "on", "on the", "the mat", "mat")
```

Слой `layer_text_vectorization()` можно настроить так, чтобы он возвращал произвольные  $N$ -граммы: биграммы, триграммы и т. д. Для этого передайте аргумент `ngrams = N`, как показано в листинге 11.7.

### Листинг 11.7 Настройка `layer_text_vectorization()` для получения биграмм

```
text_vectorization <-
  layer_text_vectorization(ngrams = 2,
                           max_tokens = 20000,
                           output_mode = "multi_hot")
```

Давайте проверим, как работает наша модель при обучении на таких мешках биграмм в двоичном коде (листинг 11.8).

### Листинг 11.8 Обучение и тестирование модели бинарных биграмм

Определим вспомогательную функцию для применения слоя `text_vectorization` к текстовому набору данных TF, потому что мы будем делать это несколько раз (с разными слоями `text_vectorization`) на протяжении всей главы

```
adapt(text_vectorization, text_only_train_ds)

dataset_vectorize <- function(dataset) {
  dataset %>%
    dataset_map(~ list(text_vectorization(.x), .y),
                num_parallel_calls = 4)
}

binary_2gram_train_ds <- train_ds %>% dataset_vectorize()
binary_2gram_val_ds <- val_ds %>% dataset_vectorize()
binary_2gram_test_ds <- test_ds %>% dataset_vectorize()

model <- get_model()
model
```

```
Model: "model_1"
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 20000)]	0
dense_3 (Dense)	(None, 16)	320016
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 1)	17
Total params: 320,033		
Trainable params: 320,033		
Non-trainable params: 0		

```
callbacks <- list(callback_model_checkpoint("binary_2gram.keras",
                                           save_best_only = TRUE))
```

```
model %>% fit(
  dataset_cache(binary_2gram_train_ds),
  validation_data = dataset_cache(binary_2gram_val_ds),
  epochs = 10,
  callbacks = callbacks
```

```
)  
  
model <- load_model_tf("binary_2gram.keras")  
evaluate(model, binary_2gram_test_ds["accuracy"]) %>%  
  sprintf("Test acc: %.3f\n", .) %>% cat()
```

| Test acc: 0.895

Теперь мы получаем точность теста 89,5 % – заметное улучшение! Оказывается, локальный порядок слов очень важен.

## БИГРАММЫ С КОДИРОВАНИЕМ TF-IDF

Вы можете добавить немного больше информации к представлению текста, подсчитав, сколько раз встречается каждое слово или  $N$ -грамма, то есть взяв гистограмму слов по тексту:

```
c("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)
```

Если вы выполняете классификацию текста, важно знать, сколько раз слово встречается в образце: любой достаточно длинный обзор фильма может содержать слово «ужасный» независимо от эмоциональной окраски, но обзор, который содержит слово «ужасный» много раз, скорее всего, отрицательный. Пример подсчета вхождений биграмм с помощью `layer_text_vectorization()` показан в листинге 11.9.

### Листинг 11.9 Настройка `layer_text_vectorization()` для получения счетчиков токенов

```
text_vectorization <-  
  layer_text_vectorization(ngrams = 2,  
                           max_tokens = 20000,  
                           output_mode = "count")
```

Разумеется, некоторые слова должны встречаться чаще, чем другие, независимо от того, о чем текст. Например, в тексте на английском языке слова «the», «a», «is» и «are» всегда будут доминировать в ваших гистограммах, заглушая другие слова, несмотря на то что они почти бесполезны для классификации. Как нам решить эту проблему?

Наверняка вы уже догадались: через нормализацию. На первый взгляд, мы могли бы нормализовать количество слов, вычтя среднее значение и разделив его на дисперсию (вычисляемую по всему набору обучающих данных). Но не все так просто. Большинство векторизованных предложений почти полностью состоят из нулей (в нашем предыдущем примере было 12 ненулевых элементов и 19 988 нулевых элементов). Такие данные называют *разреженными*. Это отличное свойство, потому что оно значительно снижает вычислительную

нагрузку и риск переобучения. Вычтя среднее из каждого признака, мы устраним разреженность. Следовательно, какую бы схему нормализации мы ни выбрали, в ней можно использовать только деление. Что же тогда применять в качестве знаменателя? Лучше всего использовать так называемую *нормализацию TF-IDF* – «частота терминов, обратная частота документов» (term frequency, inverse document frequency).

### Понятие нормализации TF-IDF

Чем чаще данный термин появляется в документе, тем важнее этот термин для понимания того, о чем этот документ. В то же время частота, с которой термин появляется во *всех* документах в вашем наборе данных, также имеет значение: термины, которые появляются почти в каждом документе (например, артикли «the» или «a»), не являются особенно информативными, в то время как термины, которые появляются только в небольшом подмножестве всех текстов (например, «Herzog»), очень своеобразны и, следовательно, важны. TF-IDF – это метрика, объединяющая эти две идеи. TF-IDF вычисляет «вес» термина, беря *частоту термина*, то есть количество появлений термина в текущем документе, и деля его на *частоту документа*, которая показывает, как часто термин встречается в наборе данных. Метрику TF-IDF можно вычислить, например, при помощи следующего кода:

```

                                Подсчет количества вхождений термина в документе
tf_idf <- function(term, document, dataset) {
  term_freq <- sum(document == term)
  doc_freqs <- sapply(dataset, function(doc) sum(doc == term))
  doc_freq <- log(1 + sum(doc_freqs))
  term_freq / doc_freq
}
                                Подсчет количества документов, в которых встречается термин

```

Нормализация TF-IDF настолько востребована, что встроена в `layer_text_vectorization()`. Все, что вам нужно сделать, – это изменить значение аргумента `output_mode` на `"tf_idf"` (листинг 11.10).

#### Листинг 11.10 Настройка `layer_text_vectorization` для применения нормализации TF-IDF

```

text_vectorization <-
  layer_text_vectorization(ngrams = 2,
                           max_tokens = 20000,
                           output_mode = "tf_idf")

```

Попробуем обучить модель с новой схемой нормализации данных.

**Листинг 11.11** Обучение и тестирование модели биграмм с нормализацией TF-IDF

Вызов adapt() будет изучать веса TF-IDF в дополнение к словарю

Мы закрепляем эту операцию только за CPU, потому что она использует операции, которые еще не поддерживает устройство GPU

```
with(tf$device("CPU"), {
  adapt(text_vectorization, text_only_train_ds)
})

tfidf_2gram_train_ds <- train_ds %>% dataset_vectorize()
tfidf_2gram_val_ds <- val_ds %>% dataset_vectorize()
tfidf_2gram_test_ds <- test_ds %>% dataset_vectorize()

model <- get_model()
model
```

Model: "model\_2"

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 20000)]	0
dense_5 (Dense)	(None, 16)	320016
dropout_2 (Dropout)	(None, 16)	0
dense_4 (Dense)	(None, 1)	17
Total params: 320,033		
Trainable params: 320,033		
Non-trainable params: 0		

```
callbacks <- list(callback_model_checkpoint("tfidf_2gram.keras",
                                           save_best_only = TRUE))

model %>% fit(
  dataset_cache(tfidf_2gram_train_ds),
  validation_data = dataset_cache(tfidf_2gram_val_ds),
  epochs = 10,
  callbacks = callbacks
)

model <- load_model_tf("tfidf_2gram.keras")
evaluate(model, tfidf_2gram_test_ds)["accuracy"] %>%
  sprintf("Test acc: %.3f", .) %>% cat("\n")
```

Test acc: 0.896

Нормализация TF-IDF дает нам точность 89,6 % в задаче классификации IMDB – в данном случае прирост на 0,1 % по сравнению с предыдущим результатом не выглядит впечатляющим. Однако во многих задачах классификации текстов с другими наборами данных при использовании TF-IDF не редкость увеличение точности на 1 %.

### Экспорт модели, обрабатывающей исходные строки

В предыдущих примерах мы выполняли стандартизацию, разбиение и индексацию текста в рамках конвейера набора данных TF. Но если мы хотим экспортировать автономную модель, независимую от этого конвейера, мы должны убедиться, что она включает в себя собственную предварительную обработку текста (в противном случае вам придется заново реализовать ее в производственной среде, что может вызвать затруднения или привести к незначительным расхождениям между обучающими и производственными данными). К счастью, сделать это не составит труда.

Просто создайте новую модель, которая повторно использует ваш слой `text_vectorization` и добавит к ней только что обученную модель:

```
inputs <- layer_input(shape = c(1), dtype = "string")
outputs <- inputs %>%
  text_vectorization() %>%
  model()
inference_model <- keras_model(inputs, outputs)
```

Один входной образец – одна строка  
 Применяем предварительную обработку  
 Применяем предварительно обученную модель  
 Создаем экземпляр сквозной модели

Полученная модель может обрабатывать пакеты исходных строк:

```
raw_text_data <- "That was an excellent movie, I loved it." %>%
  as_tensor(shape = c(-1, 1))
predictions <- inference_model(raw_text_data)
str(predictions)
```

Модель ожидает, что входные данные будут пакетом выборок,  
 то есть матрицей с одним столбцом

```
<tf.Tensor: shape=(1, 1), dtype=float32, numpy=array([[0.93249124]]),
  => dtype=float32>
cat(sprintf("%.2f percent positive\n",
  as.numeric(predictions) * 100))
```

```
93.25 percent positive
```

### 11.3.3 Обработка последовательности слов

Последние примеры ясно показывают, что порядок слов имеет значение: ручное конструирование признаков, основанных на порядке, таких как биграммы, дает хороший прирост точности. А теперь вспомним, что история глубокого обучения – это отход от ручного конструирования признаков в стремлении к тому, чтобы модели самостоятельно обучались только на основе имеющихся данных. Было бы логично вместо ручного конструирования признаков попытаться обучить модель на необработанных последовательностях слов и позволить ей самостоятельно выявить скрытые признаки. Именно этим занимаются *модели последовательностей*.

Чтобы реализовать модель последовательности, вы должны начать с представления входных выборок в виде последовательностей целочисленных индексов (одно целое число соответствует одному слову). Затем вы должны сопоставить каждое целое число с вектором, чтобы получить последовательности векторов. Наконец, вы должны передать эти последовательности векторов в модели, способные находить взаимную корреляцию признаков из соседних векторов, такие как одномерная сверточная сеть, RNN или Transformer.

Некоторое время назад, примерно в 2016–2017 годах, двунаправленные RNN (в частности, двунаправленные LSTM) считались наиболее современными решениями для моделирования последовательностей. Поскольку вы уже знакомы с этой архитектурой, именно ее мы будем использовать в наших первых примерах модели последовательности. Однако в настоящее время моделирование последовательности почти повсеместно основано на архитектуре Transformer, о которой мы вскоре расскажем. Как ни странно, одномерные свертки никогда не были популярны в NLP, хотя, по моему собственному опыту, остаточный стек одномерных сверток с разделением по глубине часто может достигать точности, сравнимой с двунаправленной LSTM, при значительном снижении вычислительных затрат.

## ПЕРВЫЙ ПРАКТИЧЕСКИЙ ПРИМЕР

Попробуем применить на практике первую модель последовательности. Начнем с подготовки наборов данных, которые возвращают целочисленные последовательности.

### Листинг 11.12 Подготовка наборов данных целочисленных последовательностей

```
max_length <- 600
max_tokens <- 20000

text_vectorization <- layer_text_vectorization(
  max_tokens = max_tokens,
  output_mode = "int",
  output_sequence_length = max_length
)

adapt(text_vectorization, text_only_train_ds)

int_train_ds <- train_ds %>% dataset_vectorize()
int_val_ds <- val_ds %>% dataset_vectorize()
int_test_ds <- test_ds %>% dataset_vectorize()
```

Чтобы сохранить приемлемый размер ввода, мы усекаем его после первых 600 слов. Это разумный выбор, поскольку средняя длина отзыва составляет 233 слова, и только 5 % отзывов длиннее 600 слов

Теперь построим модель. Самый простой способ преобразовать наши целочисленные последовательности в векторные – это прямое унитарное кодирование целых чисел (каждое измерение вектора будет представлять один возможный термин в словаре). Поверх этих унитарных векторов мы добавим простую двунаправленную LSTM.

### Листинг 11.13 Модель, работающая с последовательностями унитарных векторов

```

Кодируем целые числа в бинарные 20 000-мерные векторы
inputs <- layer_input(shape(NULL), dtype = "int64")
embedded <- inputs %>%
  tf$one_hot(depth = as.integer(max_tokens))
outputs <- embedded %>%
  bidirectional(layer_lstm(units = 32)) %>%
  layer_dropout(.5) %>%
  layer_dense(1, activation = "sigmoid")

model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = "accuracy")
model

```

Добавляем  
двунаправленную LSTM

Наконец, добавляем  
слой классификации

Подаем  
на один вход  
последовательность  
целых  
чисел

```

Model: "model_4"
-----
Layer (type)                Output Shape              Param #
-----
input_5 (InputLayer)        [(None, None)]           0
tf.one_hot (TFOpLambda)     (None, None, 20000)      0
bidirectional (Bidirectional) (None, 64)                5128448
dropout_3 (Dropout)         (None, 64)                0
dense_6 (Dense)             (None, 1)                  65
-----
Total params: 5,128,513
Trainable params: 5,128,513
Non-trainable params: 0

```

Теперь обучим нашу модель (листинг 11.14).

### Листинг 11.14 Обучение первой базовой модели последовательности

```

callbacks <- list(
  callback_model_checkpoint("one_hot_bidir_lstm.keras",
    save_best_only = TRUE))

```

Первое наблюдение: эта модель обучается очень медленно, особенно по сравнению с облегченной моделью из предыдущего раздела. Это связано с тем, что наши входные данные довольно велики: каждая входная выборка кодируется как матрица (600, 20 000) (600 слов на выборку, 20 000 возможных слов). Это 12 000 000 чисел с плавающей запятой на один обзор фильма. У двунаправленной LSTM очень много работы. Вдобавок точность модели на контрольных данных составляет всего 87 % – она не работает так же хорошо, как наша (кстати, очень быстрая!) модель бинарных униграмм.

Теперь ясно, что использовать унитарное кодирование для преобразования слов в векторы было хоть и наиболее очевидной, но не самой удачной идеей. Есть способ получше: *встраивание слов*.

### Уменьшение `batch_size` для предотвращения ошибок нехватки памяти

В зависимости от комплектации вашего компьютера и доступной оперативной памяти вашего GPU вы можете столкнуться с ошибками нехватки памяти при попытке обучить более крупную двунаправленную модель. Если это произойдет, попробуйте провести обучение с меньшим размером пакета. Вы можете передать меньшее значение аргумента `batch_size` в `text_dataset_from_directory(batch_size = )` или перекомпоновать существующий набор данных TensorFlow следующим образом:

```
int_train_ds_smaller <- int_train_ds %>%
  dataset_unbatch() %>%
  dataset_batch(16)

model %>% fit(int_train_ds_smaller, validation_data = int_val_ds,
             epochs = 10, callbacks = callbacks)

model <- load_model_tf("one_hot_bidir_lstm.keras")
sprintf("Test acc: %.3f", evaluate(model, int_test_ds)["accuracy"])

[1] "Test acc: 0.873"
```

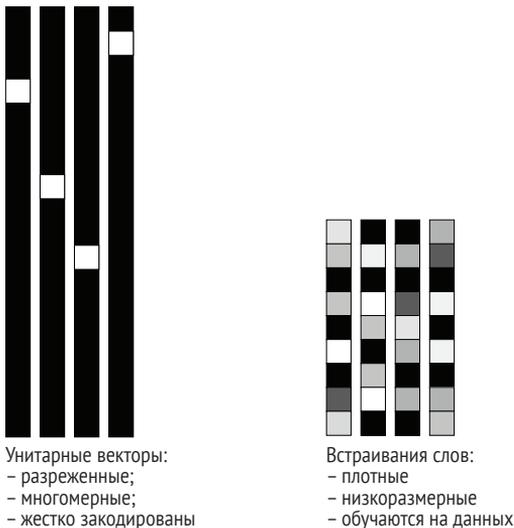
## ЧТО ТАКОЕ ВСТРАИВАНИЕ СЛОВ

Важно отметить, что, выбирая унитарное кодирование, вы принимаете инженерное решение. Вы вводите в свою модель фундаментальное предположение о структуре пространства признаков. Это предположение состоит в том, что *различные токены, которые вы кодируете, независимы друг от друга* – то есть все унитарные векторы ортогональны друг другу. Но в случае со словами это предположение явно неверно. Слова образуют структурированное пространство: они обмениваются информацией друг с другом. Слова «фильм» и «кино» взаимозаменяемы в большинстве предложений, поэтому вектор, представляющий «кино», не должен быть ортогонален вектору, представляющему «фильм», – они должны быть одним и тем же вектором или быть достаточно близки.

Чтобы стать более обобщенным, *геометрическое отношение* между двумя векторами слов должно отражать *семантическое отношение* между этими словами. Например, мы ожидаем, что в разумном векторном пространстве слов синонимы будут представлены близкими векторами слов, а геометрическое расстояние (такое как косинусное расстояние или расстояние L2) между любыми двумя векторами слов будет отражать «семантическое расстояние» между ассоциированными словами. Проще говоря, слова, несущие разные смыслы,

должны располагаться далеко друг от друга, тогда как родственные слова должны быть ближе.

*Встраивание слов* – это векторное представление слов, которое обладает важным свойством: оно отображает естественный человеческий язык в структурированное геометрическое пространство. В то время как векторы, полученные с помощью прямого унитарного кодирования, являются бинарными, разреженными (в основном состоят из нулей) и очень многомерными (такой же размерности, как количество слов в словаре), встраивания слов представляют собой векторы низкой размерности с плавающей запятой (т. е. *плотные векторы*, в отличие от разреженных векторов, рис. 11.2). При работе с очень большими словарями часто можно увидеть встраивания слов, которые имеют 256, 512 или 1024 измерения. С другой стороны, прямое унитарное кодирование слов обычно дает векторы размерностью 20 000 или более измерений (в данном случае представление словаря из 20 000 токенов). Таким образом, встраивание слов упаковывает больше информации в гораздо меньшее количество измерений.



**Рис. 11.2** Представления слов, полученные при унитарном кодировании или хешировании, являются разреженными, многомерными и жестко запрограммированными. Встраивания слов являются плотными, относительно низкоразмерными и изучаются на основе данных

Помимо того что это *плотные* представления, встраивания слов также являются структурированными представлениями, и их структура изучается из данных. Близкие по смыслу слова встраиваются в близкие позиции, и, кроме того, имеют значение определенные *направления* в пространстве встраивания. Чтобы было понятнее, давайте рассмотрим конкретный пример.

На рис. 11.3 в двумерную плоскость встроены четыре слова: *кошка*, *собака*, *волк* и *тигр*. Некоторые семантические отношения между этими словами могут быть закодированы как геометрические преобразования с учетом выбранных нами векторных представлений. Например, один и тот же вектор позволяет нам перейти от кошки к тигру и от собаки к волку: этот вектор можно интерпретировать как вектор «от домашнего животного к дикому». Точно так же другой вектор позволяет нам перейти от собаки к кошке и от волка к тигру, что можно интерпретировать как вектор «от собаки к кошке».

В реальных пространствах встраивания слов распространенными примерами осмысленных геометрических преобразований являются векторы «пола» и «множественного числа». Например, суммируя вектор «женщина» с вектором «король», мы получаем вектор «королева». Добавляя «множественный» вектор, мы получаем слово «короли». Пространства встраивания слов обычно содержат тысячи таких интерпретируемых и потенциально полезных векторов.

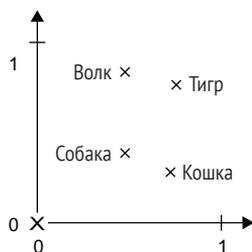


Рис. 11.3 Пример пространства встраивания слов

Далее мы рассмотрим, как использовать такое пространство встраивания на практике. Есть два способа получить встраивания слов:

- изучить встраивания слов вместе с основной задачей, которая вас интересует (например, классификация документов или предсказание эмоциональной окраски). В этой конфигурации вы начинаете со случайных векторов слов, а затем изучаете векторы слов так же, как вы изучаете веса нейронной сети;
- загрузить в свою модель готовые встраивания слов, которые были предварительно получены с использованием задачи машинного обучения, отличной от той, которую вы пытаетесь решить. Они называются *предварительно обученными встраиваниями слов*.

Разберем подробнее каждый из этих подходов.

### ИЗУЧЕНИЕ ВСТРАИВАНИЯ СЛОВ С ПОМОЩЬЮ СЛОЯ ВСТРАИВАНИЯ

Существует ли какое-то идеальное пространство встраивания слов, которое безупречно отображало бы человеческий язык и могло бы использоваться для любой задачи обработки естественного языка? Возможно, но нам еще предстоит найти что-то в этом роде. Вдоба-

вок не существует такого понятия, как *человеческий язык*, – есть много разных языков, и они не изоморфны друг другу, потому что язык – это отражение определенной культуры и определенного контекста. Но с более прагматической точки зрения требования к хорошему пространству встраивания слов во многом зависят от вашей задачи. Идеальное пространство для модели анализа настроений в обзорах фильмов может выглядеть иначе, чем идеальное пространство для модели классификации юридических документов, поскольку важность тех или иных семантических отношений зависит от задачи.

Следовательно, имеет смысл обучать новое пространство встраивания для каждой новой задачи. К счастью, это легко делается с помощью алгоритма обратного распространения, а Keras делает эту работу еще проще. Речь идет о нахождении весов слоя `layer_embedding()`.

#### Листинг 11.15 Создание экземпляра `layer_embedding`

```
embedding_layer <- layer_embedding(input_dim = max_tokens,
                                   output_dim = 256)
```

`layer_embedding()` ожидает как минимум два аргумента:  
количество возможных токенов и размерность встраивания (здесь 256)

Слой `layer_embedding()` лучше всего рассматривать как словарь, который отображает целочисленные индексы (обозначающие определенные слова) в плотные векторы. Он принимает целые числа в качестве входных данных, ищет эти целые числа во внутреннем словаре и возвращает связанные векторы. Фактически это поиск по словарю (рис. 11.4).

Индекс слова → Слой встраивания → Соответствующий вектор слова

Рис. 11.4 Слой встраивания

Слой встраивания принимает в качестве входных данных двумерный тензор целых чисел формы `(batch_size, sequence_length)`, где каждая запись представляет собой последовательность целых чисел. Затем слой возвращает трехмерный тензор чисел с плавающей запятой формы `(batch_size, sequence_length, embedding_diversity)`.

Когда вы создаете экземпляр `layer_embedding()`, его веса (внутренний словарь векторов токенов) изначально случайны, как и для любого другого слоя. Во время обучения эти векторы слов постепенно корректируются с помощью обратного распространения, структурируя пространство таким образом, чтобы его могла использовать нижестоящая модель. После обучения пространство встраивания будет иметь весьма сложную структуру, настроенную под конкретную проблему, для решения которой вы обучаете свою модель.

Давайте построим модель, включающую `layer_embedding()`, и сравним ее с другими моделями на прежней задаче.

**Листинг 11.16 Модель, использующая layer\_embedding, и обученная с нуля**

```
inputs <- layer_input(shape(NA), dtype = "int64")
embedded <- inputs %>%
  layer_embedding(input_dim = max_tokens, output_dim = 256)
outputs <- embedded %>%
  bidirectional(layer_lstm(units = 32)) %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>%
  compile(optimizer = "rmsprop",
          loss = "binary_crossentropy",
          metrics = "accuracy")
model
```

Model: "model\_5"

Layer (type)	Output Shape	Param #
input_6 (InputLayer)	[(None, None)]	0
embedding_1 (Embedding)	(None, None, 256)	5120000
bidirectional_1 (Bidirectional)	(None, 64)	73984
dropout_4 (Dropout)	(None, 64)	0
dense_7 (Dense)	(None, 1)	65
Total params: 5,194,049		
Trainable params: 5,194,049		
Non-trainable params: 0		

```
callbacks <- list(callback_model_checkpoint("embeddings_bidir_lstm.
keras",
                                           save_best_only = TRUE))
model %>%
  fit(int_train_ds,
      validation_data = int_val_ds,
      epochs = 10,
      callbacks = callbacks)
model <- load_model_tf("embeddings_bidir_lstm.keras")
evaluate(model, int_test_ds)["accuracy"] %>%
  sprintf("Test acc: %.3f\n", .) %>% cat()
```

Test acc: 0.842

Она обучается намного быстрее, чем модель с унитарным кодированием (поскольку LSTM должен обрабатывать только 256-мерные векторы вместо 20 000-мерных), а точность на контрольном наборе сопоставима (84 %). Однако мы все еще далеки от результатов нашей базовой модели биграмм. Отчасти причина в том, что модель рассматривает немного меньше данных: модель биграмм обраба-

тивала полные обзоры, тогда как наша модель последовательностей усекала последовательности после 600 слов.

## ПРИНЦИП ДОПОЛНЕНИЯ И МАСКИРОВАНИЯ

Одна из причин, по которой снижается точность модели, заключается в том, что наши входные последовательности полны нулей. Дело в том, что мы использовали опцию `output_sequence_length = max_length` в `layer_text_vectorization()` (где `max_length = 600`): предложения длиннее 600 токенов усекаются до длины 600, а предложения короче 600 токенов дополняются нулями в конце, чтобы их можно было объединить с другими последовательностями для формирования непрерывных пакетов.

Мы используем двунаправленную RNN, где два слоя RNN работают параллельно – один из них обрабатывает токены в их естественном порядке, а другой обрабатывает те же токены в обратном порядке. RNN, которая просматривает токены в их естественном порядке, выполнит свои последние итерации, видя только векторы, которые кодируют заполнение – возможно, в течение нескольких сотен итераций, если исходное предложение было коротким. Информация, хранящаяся во внутреннем состоянии RNN, будет постепенно исчезать, по мере того как она подвергается воздействию этих бессмысленных входных данных.

Нам нужен какой-то способ сообщить RNN, что она должна пропустить эти итерации. Для этого применяется *маскирование*. Слой `layer_embedding()` может генерировать «маску», соответствующую входным данным. Эта маска представляет собой тензор единиц и нулей (или логических значений TRUE/FALSE) формы `(batch_size, sequence_length)`, где вход `mask[i, t]` указывает, следует ли пропустить временной шаг `t` выборки `i` (шаг будет пропущен, если значение `mask[i, t]` равно 0 или FALSE, и обработан в противном случае).

По умолчанию эта опция не активна – вы можете включить ее, передав `mask_zero = TRUE` в `layer_embedding()`. Для формирования маски предназначен метод `calculate_mask()`:

```
embedding_layer <- layer_embedding(input_dim = 10, output_dim = 256,
                                   mask_zero = TRUE)
some_input <- rbind(c(4, 3, 2, 1, 0, 0, 0),
                  c(5, 4, 3, 2, 1, 0, 0),
                  c(2, 1, 0, 0, 0, 0, 0))
mask <- embedding_layer$compute_mask(some_input)
mask
```

```
tf.Tensor(
[[ True True True True False False False]
 [ True True True True True False False]
 [ True True False False False False False]], shape=(3, 7), dtype=bool)
```

На практике вам почти никогда не придется управлять масками вручную. Вместо этого Keras автоматически передает маску каж-

дому слою, способному ее обработать (как фрагмент метаданных, прикрепленный к последовательности, которую он представляет). Слои RNN будут использовать эту маску для пропуска замаскированных шагов. Если ваша модель возвращает всю последовательность, функция потерь также будет использовать маску для пропуска шагов в выходной последовательности. Попробуем заново обучить модель с включенной маскировкой и сравнить результаты.

**Листинг 11.17** Использование слоя встраивания с включенным маскированием

```
inputs <- layer_input(c(NA), dtype = "int64")
embedded <- inputs %>%
  layer_embedding(input_dim = max_tokens,
                 output_dim = 256,
                 mask_zero = TRUE)

outputs <- embedded %>%
  bidirectional(layer_lstm(units = 32)) %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")

model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop",
                 loss = "binary_crossentropy",
                 metrics = "accuracy")

model
```

Model: "model\_6"

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, None)]	0
embedding_3 (Embedding)	(None, None, 256)	5120000
bidirectional_2 (Bidirectional)	(None, 64)	73984
dropout_5 (Dropout)	(None, 64)	0
dense_8 (Dense)	(None, 1)	65
=====		
Total params:	5,194,049	
Trainable params:	5,194,049	
Non-trainable params:	0	

```
callbacks <- list(
  callback_model_checkpoint("embeddings_bidir_lstm_with_masking.keras",
                           save_best_only = TRUE)
)

model %>% fit(
  int_train_ds,
  validation_data = int_val_ds,
  epochs = 10,
```

```

callbacks = callbacks
)

model <- load_model_tf("embeddings_bidir_lstm_with_masking.keras")
cat(sprintf("Test acc: %.3f\n",
           evaluate(model, int_test_ds)["accuracy"]))

```

```
| Test acc: 0.880
```

На этот раз мы получаем точность на контрольных данных 88 % – небольшое, но заметное улучшение.

## ИСПОЛЬЗОВАНИЕ ПРЕДВАРИТЕЛЬНО ПОДГОТОВЛЕННЫХ ВСТРАИВАНИЙ СЛОВ

Иногда у вас будет настолько мало обучающих данных, что вы не сможете использовать их, чтобы получить встраивание приемлемого словарного запаса для конкретной задачи. В таких случаях вы можете загрузить векторы встраиваний из предварительно вычисленного пространства, которое, как вы знаете, хорошо структурировано и обладает полезными свойствами, охватывая общие аспекты языковой структуры. Использование предварительно обученных встраиваний слов в обработке естественного языка во многом похоже на использование предварительно обученных сверточных сетей в классификации изображений: у вас недостаточно данных, чтобы самостоятельно изучить действительно мощные признаки, но вы предполагаете, что признаки, которые вам нужны, являются весьма обобщенными, поэтому их можно позаимствовать из другой модели. В этом случае имеет смысл повторно использовать признаки, изученные в другой задаче.

Такие встраивания слов обычно вычисляют с использованием *статистики встречаемости* (наблюдения за тем, какие слова встречаются в предложениях или документах) и не обязательно с использованием нейронных сетей. Идея плотного низкоразмерного пространства встраивания для слов, вычисляемых без учителя, была первоначально предложена Бенжио в начале 2000-х<sup>1</sup>, но фактически начала набирать популярность в исследовательских и промышленных приложениях только после выпуска одной из самых известных и успешных схем встраивания слов – алгоритма Word2Vec (<https://code.google.com/archive/p/word2vec>), разработанного Томасом Миколовым из Google в 2013 году. Измерения пространства Word2Vec отражают определенные семантические свойства, например пол.

Вы можете загрузить различные предварительно вычисленные базы данных встраивания слов и использовать их в `layer_embed-`

<sup>1</sup> Yoshua Bengio et al., *A Neural Probabilistic Language Model*, Journal of Machine Learning Research (2003).

ding(). Среди них есть и Word2Vec. Другой популярный метод называется Global Vectors for Word Representation (глобальные векторы представления слов, GloVe, <https://nlp.stanford.edu/projects/glove>). Он разработан исследователями из Стэнфорда в 2014 году. Этот метод встраивания основан на факторизации матрицы статистики совпадения слов. Разработчики метода предоставили предварительно вычисленные встраивания для миллионов английских токенов, полученные из данных Википедии и Common Crawl.

Давайте попробуем использовать готовые встраивания GloVe в нашей модели Keras. Этот метод также подходит для встраиваний Word2Vec или любой другой базы данных встраивания слов. Мы начнем с загрузки файлов GloVe и их разбора. Затем мы загрузим векторы слов в слой Keras `layer_embedding()`, который будем использовать для построения новой модели.

Загрузим встраивания слов GloVe, предварительно вычисленные на наборе данных английской Википедии 2014 года. Это ZIP-файл размером 822 МБ, содержащий 100-мерные векторы встраивания для 400 000 слов (или токенов, не являющихся словами):

```
download.file("http://nlp.stanford.edu/data/glove.6B.zip",
             destfile = "glove.6B.zip")
zip::unzip("glove.6B.zip")
```

Теперь проанализируем разархивированный текстовый файл, чтобы построить индекс, который сопоставляет слова (строковое значение) с их векторным представлением. Поскольку структура файла представляет собой числовую матрицу с именами строк, мы сделаем это в R (листинг 11.8).

**Листинг 11.18 Разбор файла встраивания слов GloVe**

<p><code>read_table()</code> возвращает <code>data.frame</code>. Аргумент <code>col_names = FALSE</code> сообщает <code>read_table()</code>, что текстовый файл не имеет строки заголовка, а сами данные начинаются с первой строки</p>	<p>Передача <code>col_types</code> не обязательна, но это лучшая практика и хорошая защита от неожиданностей (например, если вы читаете поврежденный файл или неправильный файл). Здесь мы сообщаем <code>read_table()</code>, что первый столбец имеет тип «символ», а следующие 100 – тип «числовой»</p>
<pre>path_to_glove_file &lt;- "glove.6B.100d.txt" embedding_dim &lt;- 100  df &lt;- readr::read_table(   path_to_glove_file,   col_names = FALSE,   col_types = paste0("c", strrep("n", 100)) ) embeddings_index &lt;- as.matrix(df[, -1]) rownames(embeddings_index) &lt;- df[[1]] colnames(embeddings_index) &lt;- NULL rm(df)</pre>	
<p>Удаление временного фрейма данных <code>data.frame</code> из памяти</p>	<p>Первый столбец – это слово, а остальные 100 столбцов – числовые встраивания</p> <p>Отбрасывание имен столбцов, автоматически созданных <code>read_table()</code> (в языке R фреймы данных должны иметь имена столбцов)</p>

Так выглядит `embedding_matrix`:

```
str(embeddings_index)
num [1:400000, 1:100] -0.0382 -0.1077 -0.3398 -0.1529 -0.1897 ...
- attr(*, "dimnames")=List of 2
..$ : chr [1:400000] "the" ", " "." "of" ...
..$ : NULL
```

Следующим шагом создадим матрицу встраивания, которую вы можете загрузить в `layer_embedding()`. Это должна быть матрица формы  $(\text{max\_words}, \text{embedding\_dim})$ , где каждая запись  $i$  содержит вектор размерности `embedding_dim` для слова с индексом  $i$  (индексация происходит во время токенизации).

### Листинг 11.19 Подготовка матрицы встраивания слов GloVe

```

i – целочисленный вектор номера строки в embeddings_index,
Получить словарь, проиндексированный i – целочисленный вектор номера строки в embeddings_index,
предыдущим слоем text_vectorization который соответствует определенному слову в словаре,
и 0, если не было подходящего слова
→ vocabulary <- text_vectorization %>% get_vocabulary()
str(vocabulary)

chr [1:20000] "" "[UNK]" "the" "a" "and" "of" "to" "is" "in" "it" "i" ...

→ tokens <- head(vocabulary[-1], max_tokens)

i <- match(vocabulary, rownames(embeddings_index), ←
nomatch = 0)

embedding_matrix <- array(0, dim = c(max_tokens, embedding_dim)) ←
→ embedding_matrix[i != 0, ] <- embeddings_index[i, ] ←
str(embedding_matrix) 0 в индексах, переданных в [ для массивов R, игнорируются.
Например, (1:10)[c(1,0,2,0,3)] возвращает c(1, 2, 3)
num [1:20000, 1:100] 0 0 -0.0382 -0.2709 -0.072 ...

Заполните записи в матрице соответствующим вектором Подготовка нулевой матрицы,
слов. Номера строк строки embedding_matrix соответствуют которую мы заполним векторами GloVe
индексным позициям слов в словаре. Слова, не найденные
в индексе внедрения, будут все нулями
```

`[-1]` для удаления маркера маски «» в первой позиции. `head(max_tokens)` – это просто проверка работоспособности – мы передали те же самые `max_tokens` в `text_vectorization` ранее

Наконец, мы используем `initializer_constant()` для загрузки предварительно обученных встраиваний в `layer_embedding()`. Чтобы не нарушить предварительно обученные представления во время обучения новой модели, мы замораживаем слой передачей аргумента `trainable = FALSE`:

```
embedding_layer <- layer_embedding(
  input_dim = max_tokens,
```

```

output_dim = embedding_dim,
embeddings_initializer = initializer_constant(embedding_matrix),
trainable = FALSE,
mask_zero = TRUE
)

```

Теперь мы готовы обучить новую модель, в целом идентичную предыдущей модели, но использующую 100-мерные предварительно обученные встраивания GloVe вместо 128-мерных обученных встраиваний.

**Листинг 11.20 Модель, использующая предварительно обученный слой встраивания**

```

inputs <- layer_input(shape(NA), dtype = "int64")
embedded <- embedding_layer(inputs)
outputs <- embedded %>%
  bidirectional(layer_lstm(units = 32)) %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)

model %>% compile(optimizer = "rmsprop",
                 loss = "binary_crossentropy",
                 metrics = "accuracy")

model

```

Model: "model\_7"

Layer (type)	Output Shape	Param #	
Trainable			
input_8 (InputLayer)	[(None, None)]	0	Y
embedding_4 (Embedding)	(None, None, 100)	2000000	N
bidirectional_3 (Bidirectional)	(None, 64)	34048	Y
dropout_6 (Dropout)	(None, 64)	0	Y
dense_9 (Dense)	(None, 1)	65	Y
Total params: 2,034,113			
Trainable params: 34,113			
Non-trainable params: 2,000,000			

```

callbacks <- list(
  callback_model_checkpoint("glove_embeddings_sequence_model.keras",
                           save_best_only = TRUE)
)
model %>%
  fit(int_train_ds, validation_data = int_val_ds,
      epochs = 10, callbacks = callbacks)

model <- load_model_tf("glove_embeddings_sequence_model.keras")

```

```
cat(sprintf(
    "Test acc: %.3f\n", evaluate(model, int_test_ds)["accuracy"]))
```

```
| Test acc: 0.877
```

Как видите, в этой конкретной задаче предварительно обученные встраивания не очень полезны, потому что набор данных содержит достаточно образцов для изучения специализированного пространства встраивания с нуля. Однако использование предварительно обученных встраиваний может быть очень полезным, когда вы работаете с небольшим набором данных.

## 11.4 Архитектура Transformer

Начиная с 2017 года рекуррентные нейронные сети в большинстве задач обработки естественного языка уступили первенство новой архитектуре Transformer. Сети-трансформеры были впервые представлены в статье Васвани<sup>1</sup> «Все, что вам нужно, – это внимание». Суть статьи ясна из названия: как оказалось, простой механизм под названием «нейронное внимание» можно использовать для создания мощных моделей последовательностей, в которых нет рекуррентных или сверточных слоев.

Можно без преувеличения сказать, что это открытие произвело революцию не только в обработке естественного языка. Нейронное внимание быстро стало одной из самых влиятельных идей в глубоком обучении. В этом разделе вы узнаете, как оно работает и почему оказалось настолько эффективным при работе с последовательностями. Затем мы воспользуемся механизмом самовнимания для создания кодировщика Transformer, одного из основных компонентов одноименной архитектуры, и применим его к задаче классификации обзоров фильмов на IMDB.

### 11.4.1 Механизм самовнимания

Просматривая эту книгу, вы можете пролистывать одни главы и внимательно читать другие, в зависимости от ваших целей или интересов. Что, если ваши модели будут вести себя так же? Такова простая, но мощная идея нейронного внимания: не вся входная информация, которую видит модель, одинаково важна для поставленной задачи, поэтому модели должны уделять больше внимания одним признакам и меньше – другим. Если этот подход показался вам знакомым, вы не ошиблись, потому что дважды встречали схожие механизмы в этой книге:

---

<sup>1</sup> Ashish Vaswani et al., *Attention Is All You Need* (2017), <https://arxiv.org/abs/1706.03762>.

- объединение по максимальным элементам из соседних просматривает пул элементов в пространственном регионе и выбирает только один из них для сохранения. Это внимание по принципу «все или ничего»: сохраняем самый важный признак и отказываемся от остальных;
- нормализация TF-IDF присваивает оценки важности токенам в зависимости от того, сколько информации они могут нести. Важные токены усиливаются, а ненужные исчезают. Это разновидность постоянного внимания.

Вы можете придумать множество различных форм внимания, но все они начинаются с оценки важности набора признаков, при этом чем выше релевантность признака задаче, тем выше его оценка (рис. 11.5). Как эти оценки должны быть рассчитаны и что вы должны с ними делать, зависит от конкретного механизма внимания.

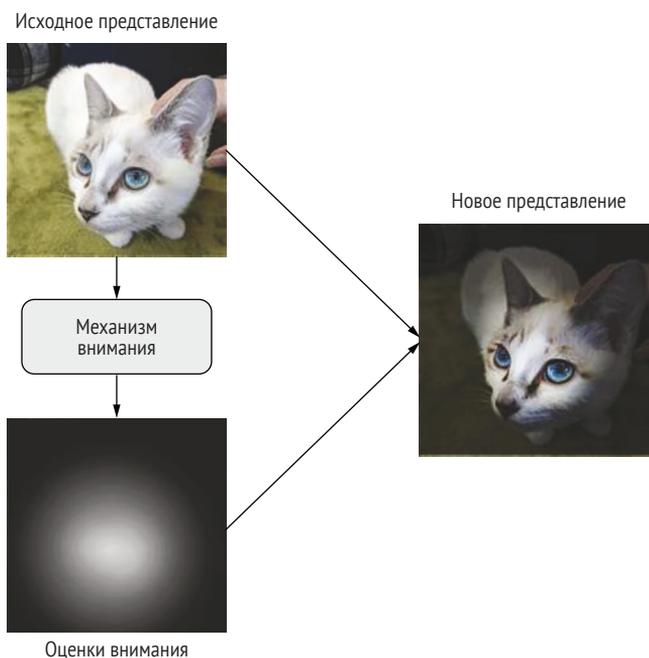
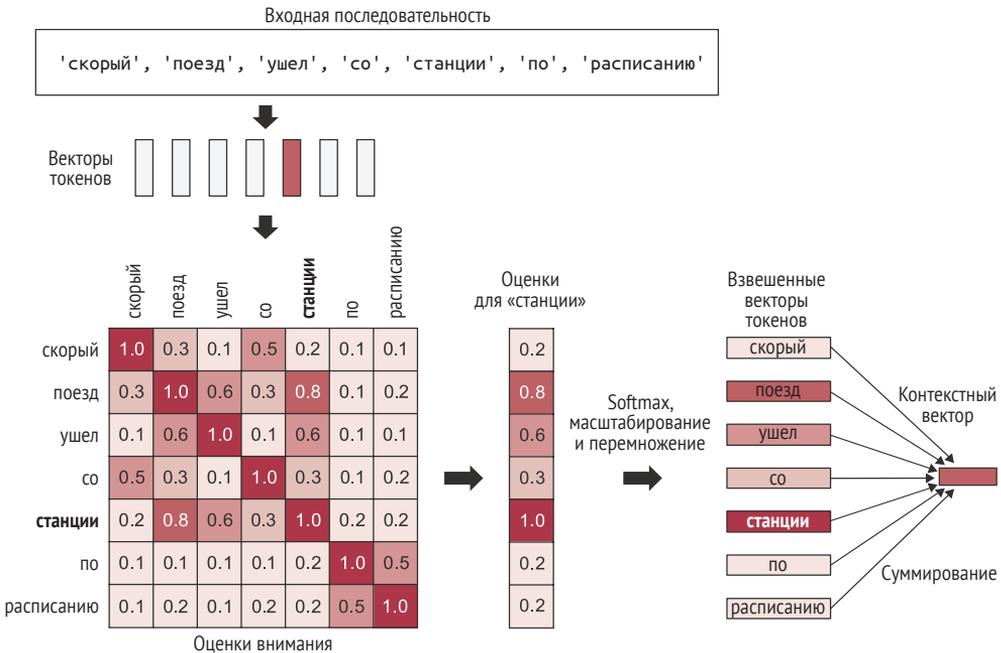


Рис. 11.5 Общая концепция внимания в глубоком обучении. Входным признакам присваиваются «оценки внимания», которые можно использовать для информирования следующего представления входных данных

Важно отметить, что такой механизм внимания можно использовать не только для выделения или затенения определенных признаков. Его можно использовать, чтобы сделать признаки *контекстно-зависимыми*. В предыдущем разделе вы узнали о встраивании слов – векторном пространстве, которое хранит форму семантических отношений между разными словами. В многомерном пространстве встраиваний слово имеет определенную позицию – фиксирован-

ный набор отношений с каждым другим словом в пространстве. Но естественный язык работает не совсем так: значение слова обычно зависит от контекста. «Свидетель на свадьбе» и «свидетель преступления» – это разные понятия. Когда вы говорите: «Мне сегодня везет», значение слова «везет» сильно отличается от такого же слова в предложении «Лошадь везет дрова». И конечно же, значение таких местоимений, как «он», «оно», «ты» и т. д., полностью зависит от предложения и даже может меняться несколько раз в одном предложении.

Иными словами, интеллектуальное пространство встраивания обеспечивает различное векторное представление слова в зависимости от окружающих его слов. Вот где вступает в действие *самовнимание*. Цель самовнимания – менять представление токена, используя представления связанных токенов в последовательности. Это создает контекстно-зависимые представления токенов. Рассмотрим предложение «Скорый поезд ушел со станции по расписанию». Вроде бы все понятно. Теперь возьмем только слово «станция». О какой станции идет речь? Может, это радиостанция? Может, Международная космическая станция? Давайте посмотрим, как решает эту проблему механизм самовнимания (рис. 11.6).



**Рис. 11.6** Механизм самовнимания. Показатели внимания рассчитываются между словом «станция» и каждым другим словом в последовательности, а затем они используются для взвешивания суммы векторов слов, которая становится новым вектором «станции»

Шаг 1 заключается в вычислении показателей релевантности между вектором слова «станция» и каждым другим словом в предло-

жении. Это наши оценки внимания. Мы будем использовать скалярное произведение двух векторов слов в качестве меры их взаимосвязи. Это очень эффективная с вычислительной точки зрения функция расстояния, которая применялась для оценки взаимосвязи двух встраиваний слов задолго до появления архитектуры Transformer. На практике эти оценки также будут проходить через функцию масштабирования и softmax, но пока это просто деталь реализации.

Шаг 2 – вычислить сумму всех векторов слов в предложении, взвешенную по нашим показателям релевантности. Слова, тесно связанные со словом «станция», будут вносить больший вклад в сумму (включая само слово «станция»), тогда как нерелевантные слова почти ничего не дадут. Результирующий вектор – это наше новое представление для «станции», которое включает в себя окружающий контекст. В частности, он содержит часть вектора «поезд», уточняя, что в данном случае «станция» фактически означает «вокзал».

Можно повторить этот процесс для каждого слова в предложении, создав новую последовательность векторов, кодирующих предложение. Обобщенный пример реализации алгоритма представлен в следующем псевдокоде:

```
self_attention <- function(input_sequence) {
  c(sequence_len, embedding_size) %<-% dim(input_sequence)

  %% с двумя одномерными векторами возвращает скаляр. Оценки имеют форму (sequence_len)
  output <- array(0, dim(input_sequence))
  for (i in 1:sequence_len) {
    pivot_vector <- input_sequence[i, ]
    scores <- sapply(1:sequence_len, function(j)
      pivot_vector %%% input_sequence[j, ])
    scores <- softmax(scores / sqrt(embedding_size))
    broadcast_scores <- as.matrix(scores)[, rep(1, embedding_size)]
    new_pivot_representation <- colSums(input_sequence * broadcast_scores)
    output[i, ] <- new_pivot_representation
  }
  output
}

softmax <- function(x) {
  e <- exp(x - max(x))
  e / sum(e)
}
```

Итерация по каждому токenu в последовательности

Вычисление скалярного произведения (показателя внимания) между текущим и любым другим токеном

Масштабирование с помощью коэффициента нормализации и применение softmax

Суммирование входных последовательностей с поправкой на оценку, чтобы создать новый вектор встраивания

Перевод вектора результатов (shape: (sequence\_len)) в матрицу формы (sequence\_len, embedding\_size), т. е. форму input\_sequence

Конечно, на практике вы бы использовали векторизованную реализацию. Keras имеет встроенный слой `layer_multi_head_attention()`. Вот как вы можете его использовать:

```
num_heads <- 4
embed_dim <- 256
mha_layer <- layer_multi_head_attention(num_heads = num_heads,
                                        key_dim = embed_dim)
outputs <- mha_layer(inputs, inputs, inputs)
```

Вход имеет форму (batch\_size, sequence\_length, embed\_dim) ←

Вероятно, у вас возникли вопросы:

- почему мы передаем входные данные слою три раза? Это кажется излишним;
- о каком *многоголовом внимании* (multi head attention) идет речь? Звучит пугающе – эти головы тоже вырастают, если их отрубить?

На оба этих вопроса есть простые ответы.

## ОБОБЩЕННОЕ САМОВНИМАНИЕ:

### МОДЕЛЬ «ЗАПРОС–КЛЮЧ–ЗНАЧЕНИЕ»

До сих пор мы рассматривали только одну входную последовательность. Однако архитектура Transformer изначально была разработана для машинного перевода, где вам приходится иметь дело с двумя входными последовательностями: исходной последовательностью, которую вы переводите в данный момент (например, «Какая сегодня погода?»), и целевой последовательностью, которую вы хотите получить (например, «¿Qué tiempo hace hoy?»). Transformer – это модель типа «последовательность–последовательность»: она была разработана специально для преобразования одной последовательности в другую. Далее в этой главе я подробно расскажу о таких моделях.

А сейчас для лучшего понимания сделаем шаг назад. Механизм самовнимания, как мы его представили, схематично работает следующим образом:

```
outputs <- sum(inputs * pairwise_scores(inputs, inputs))
```

↑  
C

↑    ↑  
A     B

Это означает «для каждого токена входных данных `inputs` (A) вычислить, насколько этот токен связан с каждым токеном во входных данных `inputs` (B), и использовать эти оценки для взвешивания суммы токенов во входных данных `inputs` (C)». При этом вовсе не обязательно, чтобы A, B и C ссылались на одну и ту же входную последовательность. В общем случае мы можем сделать это с тремя разными последовательностями. Мы будем называть их «запрос», «ключи» и «значения». Теперь у нас получается следующий процесс: «для каждого элемента в запросе вычислить, насколько элемент свя-

зан с каждым ключом, и использовать эти оценки для взвешивания суммы значений»:

```
outputs <- sum(values * pairwise _ scores(query, keys))
```

Эта терминология позаимствована из поисковых и рекомендательных систем (рис. 11.7). Представьте, что вы вводите запрос для поиска фотографии из вашей коллекции «собаки на пляже». Каждая из ваших фотографий в базе данных описывается набором ключевых слов – «кошка», «собака», «вечеринка» и т. д. Мы будем называть их «ключами». Поисковая система начинает со сравнения вашего запроса с ключами в базе данных. «Собака» дает совпадение 1, а «кошка» дает совпадение 0. Затем поисковая система ранжирует ключи по степени совпадения с запросом – *релевантности* – и возвращает изображения, связанные с лучшими  $N$  совпадениями, в порядке убывания релевантности.

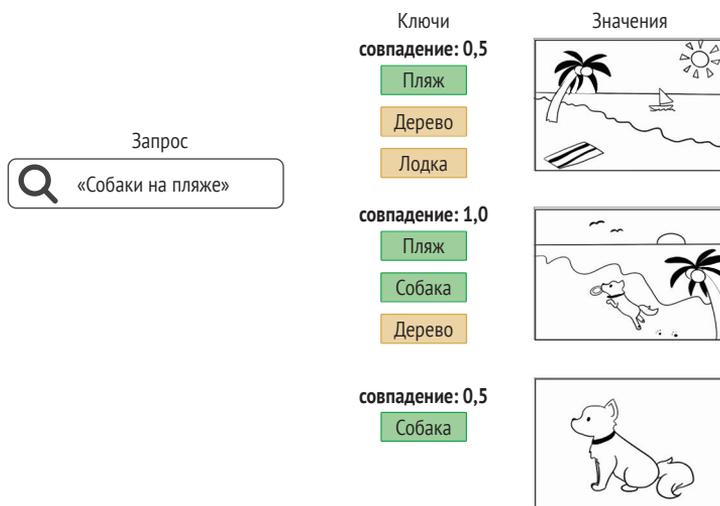


Рис. 11.7 Извлечение изображений из базы данных. «Запрос» сравнивается с набором «ключей», а оценки соответствия используются для ранжирования «значений» (изображений)

Именно по такому принципу работает внимание в стиле Transformer. У вас есть эталонная последовательность, описывающая то, что вы ищете: запрос. У вас также есть совокупность знаний, из которых вы пытаетесь извлечь информацию: значения. Каждому значению назначается ключ, описывающий значение в формате, который можно легко сравнить с запросом. Вы просто сопоставляете запрос с ключами. Затем вы возвращаете взвешенную сумму значений.

На практике ключи и значения часто представляют собой одинаковую последовательность. Например, в машинном переводе запрос будет целевой последовательностью, а исходная последовательность

будет играть роли как ключей, так и значений: для каждого элемента цели (например, «tiempo») вы должны вернуться к источнику («Какая сегодня погода?») и определить различные элементы, связанные с ним («tiempo» и «погода» должны иметь точное соответствие). И разумеется, если вы просто классифицируете последовательность, то запрос, ключи и значения одинаковы: вы сравниваете последовательность с самой собой, чтобы обогатить каждый токен контекстом из всей последовательности.

Это объясняет, почему нам нужно было трижды передать входные данные нашему слою `layer_multi_head_attention()`. Но почему внимание называют «многоголовым»?

### 11.4.2 Многоголовое внимание

*Многоголовое внимание* – это расширенная версия механизма самовнимания, представленного в статье «Все, что вам нужно, – это внимание». Эпитет «многоголовый» обозначает тот факт, что выходное пространство слоя самовнимания разбивается на набор независимых подпространств, изучаемых отдельно: исходный запрос, ключ и значение отправляются через три независимых набора плотных проекций, что дает нам три отдельных вектора. Каждый вектор обрабатывается с помощью нейронного внимания, и три выхода объединяются обратно в единую выходную последовательность. Каждое такое подпространство называется «головой». Полная картина показана на рис. 11.8.

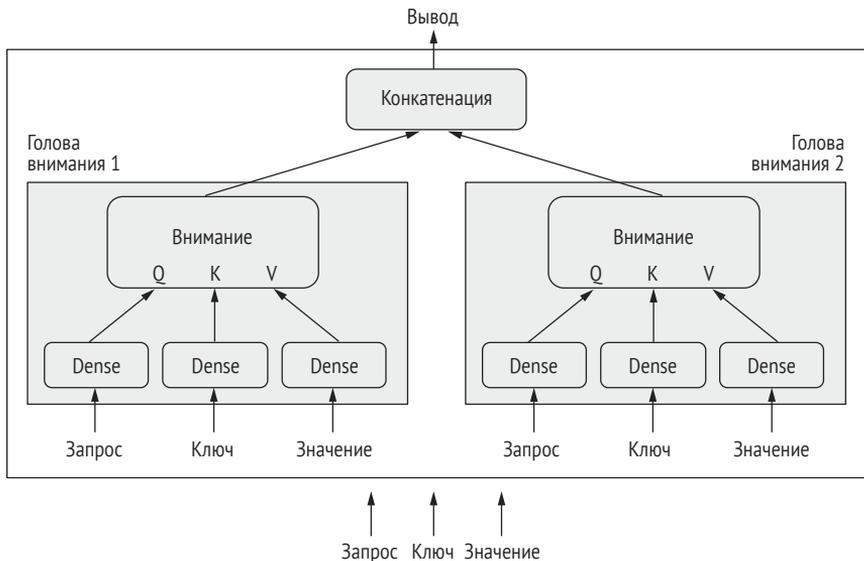


Рис. 11.8 Слой многоголового внимания `MultiHeadAttention`

Наличие обучаемых плотных проекций позволяет слою действительно чему-то научиться, в отличие от преобразования без сохранения состояния, которому потребуются дополнительные слои до или после него, чтобы быть полезным. Кроме того, наличие независимых голов помогает слою изучать разные группы признаков для каждого токена, когда признаки в одной группе коррелируют друг с другом, но в основном независимы от признаков в другой группе.

В принципе, это похоже на работу механизма свертки с разделением по глубине: в ней выходное пространство свертки разбивается на множество подпространств (по одному на входной канал), которые изучаются независимо. Статья про внимание, на которую я постоянно ссылаюсь, была написана в то время, когда стало ясно, что идея разложения пространств признаков на независимые подпространства обеспечивает большие преимущества для моделей компьютерного зрения как в случае сверток с разделением по глубине, так и в случае *сгруппированных сверток*. Многоголовое внимание – это просто применение той же идеи к самовниманию.

### 11.4.3 Кодировщик в архитектуре Transformer

Если добавление дополнительных плотных проекций приносит такую пользу, почему бы нам не применить одну или две к выходу механизма внимания? Согласен, это отличная идея – давайте скорее этим займемся. Наша модель становится все сложнее, поэтому имеет смысл добавить остаточные связи, чтобы избежать затухания ценной информации на длинном пути; в главе 9 вы узнали, что остаточные связи необходимы для любой достаточно глубокой архитектуры. И есть еще одна вещь, которую вы узнали в главе 9: предполагается, что слои нормализации помогают градиентному спуску во время обратного распространения. Добавим и их в общий котел.

Примерно такой мыслительный процесс, как мне кажется, происходил в умах изобретателей архитектуры Transformer. Распределение выходных данных по нескольким независимым пространствам, добавление остаточных соединений, добавление уровней нормализации – все это стандартные архитектурные шаблоны, которые было бы целесообразно использовать в любой сложной модели. Вместе эти компоненты образуют *кодировщик* – одну из двух важнейших составных частей архитектуры Transformer (рис. 11.9).

Базовая архитектура Transformer состоит из двух частей: кодировщика, который обрабатывает исходную последовательность, и декодера, который использует исходную последовательность для создания переведенной версии. Немного позже я расскажу о декодере.

Важно отметить, что кодировщик можно использовать для классификации текста. Это очень обобщенный модуль, который принимает последовательность и учится превращать ее в более полезное представление. Давайте разработаем кодировщик Transformer

и протестируем его на задаче классификации эмоциональной окраски обзоров фильмов.

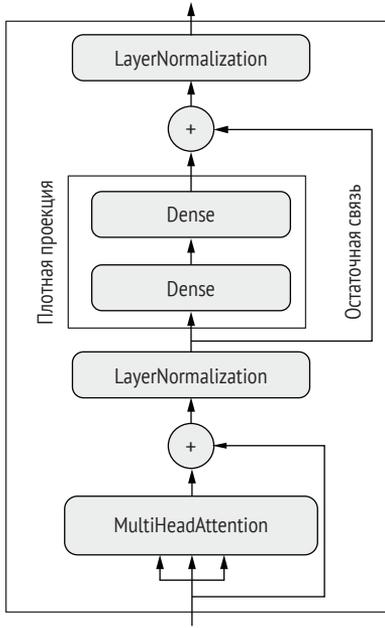


Рис. 11.9 TransformerEncoder связывает layer\_multi\_head\_attention() с плотной проекцией и добавляет нормализацию, а также остаточные связи

**Листинг 11.21 Кодер Transformer, реализованный как подкласс Layer**

```

layer_transformer_encoder <- new_layer_class(
  classname = "TransformerEncoder",
  initialize = function(embed_dim, dense_dim, num_heads, ...) {
    super$initialize(...)
    self$embed_dim <- embed_dim
    self$dense_dim <- dense_dim
    self$num_heads <- num_heads
    self$attention <-
      layer_multi_head_attention(num_heads = num_heads,
                                key_dim = embed_dim)

    self$dense_proj <- keras_model_sequential() %>%
      layer_dense(dense_dim, activation = "relu") %>%
      layer_dense(embed_dim)

    self$layernorm_1 <- layer_layer_normalization()
    self$layernorm_2 <- layer_layer_normalization()
  },
  call = function(inputs, mask = NULL) {
    if (!is.null(mask))
      mask <- mask[, tf$newaxis, ]
  }
  )
  
```

Вычисления выполняются в call()

Маска, которая будет сгенерирована слоем встраивания, будет двумерной, но слой внимания ожидает, что она будет трех- или четырехмерной, поэтому мы расширяем ее размерность

```

Выполняем      inputs %>%
сериализацию,  { self$attention(., ., attention_mask = mask) + . } %>% ←
благодаря чему self$layernorm_1(.) %>%
можно сохранить модель { self$dense_proj(.) + . } %>% ←
                                     Добавляем остаточную
                                     связь к выходу слоя
                                     dense_proj()
                                     },
                                     Добавляем
                                     остаточную связь
                                     к выходу слоя
                                     внимания
get_config = function() {
  config <- super$get_config()
  for(name in c("embed_dim", "num_heads", "dense_dim"))
    config[[name]] <- self[[name]]
  config
}
)

```

### Операторы %>% и { }

В приведенном выше примере мы передаем аргумент при помощи %>% в выражение, заключенное в фигурные скобки {}. Это особая функция оператора %>%, которая позволяет вам передавать сложные или составные выражения. Через канал %>% аргумент будет подставлен в каждое местоположение, обозначенное с помощью символа точки. Например, строка кода

```
x %>% { fn(., .) + . }
```

эквивалентна строке

```
fn(x, x) + x
```

Если бы мы написали метод call() для layer\_transformer\_encoder() без использования %>%, он бы выглядел так:

```

call = function(inputs, mask = NULL) {
  if (!is.null(mask))
    mask <- mask[, tf$newaxis, ]
  attention_output <- self$attention(inputs, inputs,
                                     attention_mask = mask)
  proj_input <- self$layernorm_1(inputs + attention_output)
  proj_output <- self$dense_proj(proj_input)
  self$layernorm_2(proj_input + proj_output)
}

```

### Сохранение пользовательских слоев

Когда вы пишете пользовательские слои, обязательно реализуйте метод get\_config(): это позволяет восстановить слой из его конфигурации, что полезно при сохранении и загрузке модели. Метод должен возвращать именованный список R, содержащий значения аргументов конструктора, используемых для создания слоя.

Все слои Keras можно сериализовать и десериализовать следующим образом:

```
config <- layer$get_config()
new_layer <- do.call(layer_<type>, config)
```

где `layer_<type>` – исходный конструктор слоя. Например:

```
layer <- layer_dense(units = 10)
config <- layer$get_config()
new_layer <- do.call(layer_dense, config)
```

Конфигурация не содержит значений весов,  
поэтому все веса в слое инициализируются с нуля

`config` – это обычный именованный список R. Вы можете безопасно сохранить его на диск как `rds`, а затем загрузить в новом сеансе R

Вы также можете получить доступ к развернутому исходному конструктору слоя из любого существующего слоя напрямую через специальный символ `__class__` (хотя в этом редко возникает необходимость):

```
layer$`__class__`
<class 'keras.layers.core.dense.Dense'>
```

```
new_layer <- layer$`__class__`$from_config(config)
```

Определение метода `get_config()` в классах пользовательских слоев позволяет использовать один и тот же рабочий процесс. Например:

```
layer <- layer_transformer_encoder(embed_dim = 256, dense_dim = 32,
                                  num_heads = 2)
config <- layer$get_config()
new_layer <- do.call(layer_transformer_encoder, config)
# -- or --
new_layer <- layer$`__class__`$from_config(config)
```

При сохранении модели, содержащей пользовательские слои, сохраненный файл будет содержать эти конфигурации. При загрузке модели из файла вы должны предоставить классы пользовательских слоев процессу загрузки, чтобы он мог понять объекты конфигурации:

```
model <- save_model_tf(model, filename)
model <- load_model_tf(filename,
                      custom_objects = list(layer_transformer_encoder))
```

Учтите, что если список, предоставленный в `custom_objects`, является именованным, то имена сопоставляются с аргументом `classname`, который был предоставлен при создании пользовательского объекта:

```
model <- load_model_tf(
  filename,
  custom_objects = list(TransformerEncoder = layer_transformer_encoder))
```

Вы могли заметить, что слои нормализации, которые мы здесь используем, не являются слоями `layer_batch_normalization()`, как те, которые мы использовали ранее в моделях изображений. Это связано с тем, что `layer_batch_normalization()` плохо работает с данными последовательностей. Вместо этого мы используем `layer_layer_normalization()`, который нормализует каждую последовательность независимо от других последовательностей в пакете. Так выглядит псевдокод в стиле R:

```
layer_normalization <- function(batch_of_sequences) {
  c(batch_size, sequence_length, embedding_dim) %<-%
  dim(batch_of_sequences)
  means <- variances <-
  array(0, dim = dim(batch_of_sequences))
  for (b in seq(batch_size))
    for (s in seq(sequence_length)) {
      embedding <- batch_of_sequences[b, s, ]
      means[b, s, ] <- mean(embedding)
      variances[b, s, ] <- var(embedding)
    }
  (batch_of_sequences - means) / variances
}
```

← Форма входа (batch\_size, sequence\_length, embedding\_dim)

← Чтобы вычислить среднее значение и дисперсию, мы объединяем данные только по последней оси (ось -1, ось встраивания)

Сравните его с `layer_batch_normalization()` (во время обучения):

```
batch_normalization <- function(batch_of_images) {
  c(batch_size, height, width, channels) %<-%
  dim(batch_of_images)
  means <- variances <-
  array(0, dim = dim(batch_of_images))
  for (ch in seq(channels)) {
    channel <- batch_of_images[, , , ch]
    means[, , , ch] <- mean(channel)
    variances[, , , ch] <- var(channel)
  }
  (batch_of_images - means) / variances
}
```

← Форма входа (batch\_size, height, width, channels)

← Объединение данных по оси пакета (первой оси), что создает взаимодействие между образцами в пакете

Несмотря на то что `batch_normalization()` собирает информацию из множества выборок для получения точных статистических данных о средних и дисперсиях признаков, `layer_normalization()` объединяет данные в каждой последовательности отдельно, что больше подходит для данных в последовательности.

Теперь, когда у нас есть собственный `TransformerEncoder`, мы можем использовать его для сборки модели классификации текста, аналогичной модели на основе LSTM, которую мы протестировали ранее.

### Листинг 11.22 Использование Transformer Encoder для классификации текста

```

vocab_size <- 20000
embed_dim <- 256
num_heads <- 2
dense_dim <- 32

inputs <- layer_input(shape(NA), dtype = "int64")
outputs <- inputs %>%
  layer_embedding(vocab_size, embed_dim) %>%
  layer_transformer_encoder(embed_dim, dense_dim, num_heads) %>%
  layer_global_average_pooling_1d() %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop",
                 loss = "binary_crossentropy",
                 metrics = "accuracy")
model

```

Поскольку TransformerEncoder возвращает полные последовательности, нам нужно сократить каждую последовательность до одного вектора для классификации через слой глобального объединения

```

Model: "model_8"

```

Layer (type)	Output Shape	Param #
input_10 (InputLayer)	[(None, None)]	0
embedding_5 (Embedding)	(None, None, 256)	5120000
transformer_encoder_1 (TransformerEncoder)	(Transform (None, None, 256))	543776
global_average_pooling1d (AveragePooling1D)	(Global (None, 256))	0
dropout_7 (Dropout)	(None, 256)	0
dense_17 (Dense)	(None, 1)	257

```

Total params: 5,664,033
Trainable params: 5,664,033
Non-trainable params: 0

```

После обучения модели точность на контрольных данных достигает 88,5 %.

### Листинг 11.23 Обучение и оценка модели на основе кодировщика Transformer

```

callbacks = list(callback_model_checkpoint("transformer_encoder.keras",
                                          save_best_only = TRUE))

model %>% fit(
  int_train_ds,
  validation_data = int_val_ds,
  epochs = 20,
  callbacks = callbacks

```

```

)
model <- load_model_tf(
  "transformer_encoder.keras",
  custom_objects = layer_transformer_encoder)
sprintf("Test acc: %.3f", evaluate(model, int_test_ds)["accuracy"])
[1] "Test acc: 0.885"

```

Предоставляем пользовательский  
класс TransformerEncoder  
процессу загрузки модели

В этот момент вы должны начать чувствовать себя немного неловко. Что-то здесь не так. Догадались ли вы, в чем дело?

Эта часть главы якобы посвящена моделям последовательности. Я начал с того, что подчеркнул важность порядка слов. Я сказал, что Transformer – это архитектура обработки последовательности, изначально разработанная для машинного перевода. И тем не менее ... кодировщик Transformer, который вы только что видели в действии, вовсе не был моделью последовательности. Он состоит из плотных слоев, которые обрабатывают токены последовательности независимо друг от друга, и уровня внимания, который рассматривает токены как набор. Вы можете изменить порядок токенов в последовательности, и вы получите точно такие же попарные оценки внимания и точно такие же контекстно-зависимые представления. Если бы вы полностью перепутали слова в каждом обзоре фильма, модель этого бы не заметила, и вы все равно получили бы прежнюю точность. Самовнимание – это механизм обработки данных, сосредоточенный на отношениях между парами элементов последовательности (см. рис. 11.10), – он слеп к тому, встречаются ли эти элементы в начале, в конце или в середине последовательности. Так почему же мы говорим, что Transformer – это модель последовательности? И как подобная модель улучшает качество машинного перевода, если она не учитывает порядок слов?

Я намекнул на ответ ранее в этой главе, когда мимоходом упомянул, что Transformer представляет собой гибридный подход, который технически не зависит от порядка, но который принудительно вводит информацию о порядке в обрабатываемые им представления. Это и есть недостающий ингредиент, который называется *позиционным кодированием*. Взгляните на табл. 11.1.

**Таблица 11.1. Характерные особенности различных типов моделей НЛП**

	Зависимость от порядка слов	Зависимость от контекста (взаимодействие между словами)
<b>Мешок униграмм</b>	Нет	Нет
<b>Мешок биграмм</b>	Ограниченная	Нет
<b>RNN</b>	Да	Нет
<b>Самовнимание</b>	Нет	Да
<b>Transformer</b>	Да	Да

## ИСПОЛЬЗОВАНИЕ ПОЗИЦИОННОГО КОДИРОВАНИЯ ДЛЯ ДОБАВЛЕНИЯ ИНФОРМАЦИИ О ПОРЯДКЕ СЛОВ

Идея позиционного кодирования очень проста: чтобы предоставить модели информацию о порядке слов, достаточно добавить позицию слова в предложении к каждому встраиванию слова. Таким образом, встраивания входных слов будут иметь два компонента: обычный вектор слова, который представляет слово независимо от какого-либо конкретного контекста, и вектор позиции, который представляет положение слова в текущем предложении. Остается надеяться, что в процессе обучения модель сама поймет, как лучше всего использовать эту дополнительную информацию.

Самая простая схема, которую только можно придумать, – это соединить позицию слова с его вектором встраивания. Можно добавить к вектору ось позиции и подставить в нее 0 для первого слова в последовательности, 1 для второго и т. д. Однако это не идеальное решение, потому что позиции потенциально могут быть слишком большими целыми числами, которые выйдут за диапазон значений в векторе встраивания. Как вы знаете, нейронные сети не любят очень большие входные значения или дискретные входные распределения.

В исходной статье про внимание предложен интересный прием для кодирования позиций слов: к встраиваниям слов добавлялся вектор, содержащий значения в диапазоне  $[-1, 1]$ , которые циклически менялись в зависимости от позиции (использовались косинусные функции). Этот прием позволяет однозначно охарактеризовать любое целое число в большом диапазоне с помощью вектора малых значений, но мы поступим еще проще и эффективнее: изучим векторы позиционного встраивания точно так же, как мы учимся встраивать индексы слов. Затем добавим позиционные встраивания к соответствующим встраиваниям слов, чтобы внести информацию о позиции. Этот метод так и называется – *позиционное встраивание* (positional embedding). Давайте реализуем его в коде.

### Листинг 11.24 Реализация позиционного встраивания в виде подкласса

```
layer_positional_embedding <- new_layer_class(
  classname = "PositionalEmbedding",

  initialize = function(sequence_length, ←
                        input_dim, output_dim, ...) {
    super$initialize(...)
    self$token_embeddings <-
      layer_embedding(input_dim = input_dim,
                     output_dim = output_dim)
```

Подготовка `layer_embedding()`  
для индексов токенов

Недостатком позиционного встраивания является то, что  
длина последовательности должна быть известна заранее

```

Подготовка layer_embedding() для позиций токенов
self$position_embeddings <-
  layer_embedding(input_dim = sequence_length,
                 output_dim = output_dim)
self$sequence_length <- sequence_length
self$input_dim <- input_dim
self$output_dim <- output_dim
},

call = function(inputs) {
  len <- tf$shape(inputs)[-1]
  positions <-
    tf$range(start = 0L, limit = len, delta = 1L)
  embedded_tokens <- self$token_embeddings(inputs)
  embedded_positions <- self$position_embeddings(positions)
  embedded_tokens + embedded_positions
},

compute_mask = function(inputs, mask = NULL) {
  inputs != 0
},

get_config = function() {
  config <- super$get_config()
  for(name in c("output_dim", "sequence_length", "input_dim"))
    config[[name]] <- self[[name]]
  config
}
)

```

tf\$shape(inputs)[-1] вырезает последний элемент формы (tf\$shape() возвращает форму в виде тензора)

tf\$range() похож на seq() в R, создает целочисленную последовательность: [0, 1, 2, ..., limit - 1]

Как и layer\_embedding(), этот слой должен уметь генерировать маску, чтобы мы могли игнорировать заполнение нулями во входных данных. Метод calculate\_mask() будет автоматически вызван фреймворком, и маска распространится на следующий уровень

Складываем векторы встраивания

Сериализация для сохранения модели

Вы должны использовать `layer_positional_embedding()` так же, как обычный `layer_embedding()`. Давайте посмотрим на него в действии!

## Соединяем компоненты: TRANSFORMER для классификации ТЕКСТА

Все, что вам нужно сделать, чтобы начать учитывать порядок слов, – это заменить старый `layer_embedding()` на новую версию с учетом позиции.

### Листинг 11.25 Объединение кодировщика Transformer с позиционным встраиванием

```

vocab_size <- 20000
sequence_length <- 600
embed_dim <- 256
num_heads <- 2
dense_dim <- 32

inputs <- layer_input(shape(NULL), dtype = "int64")

```

```

outputs <- inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  layer_transformer_encoder(embed_dim, dense_dim, num_heads) %>%
  layer_global_average_pooling_1d() %>%
  layer_dropout(0.5) %>%
  layer_dense(1, activation = "sigmoid")

model <-
  keras_model(inputs, outputs) %>%
  compile(optimizer = "rmsprop",
          loss = "binary_crossentropy",
          metrics = "accuracy")

```

```
model
```

```
Model: "model_9"
```

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, None)]	0
positional_embedding (Positional Embedding)	(None, None, 256)	5273600
transformer_encoder_2 (TransformerEncoder)	(None, None, 256)	543776
global_average_pooling1d_1 (GlobalAveragePooling1D)	(None, 256)	0
dropout_8 (Dropout)	(None, 256)	0
dense_22 (Dense)	(None, 1)	257

```
Total params: 5,817,633
```

```
Trainable params: 5,817,633
```

```
Non-trainable params: 0
```

```

callbacks <- list(
  callback_model_checkpoint("full_transformer_encoder.keras",
                            save_best_only = TRUE)
)

model %>% fit(
  int_train_ds,
  validation_data = int_val_ds,
  epochs = 20,
  callbacks = callbacks
)

model <- load_model_tf(
  "full_transformer_encoder.keras",
  custom_objects = list(layer_transformer_encoder,
                        layer_positional_embedding))

cat(sprintf(
  "Test acc: %.3f\n", evaluate(model, int_test_ds)["accuracy"]))

```

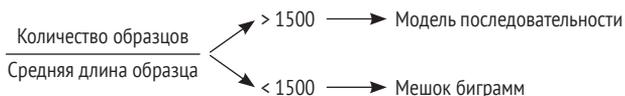
```
Test acc: 0.886
```

Смотрите-ка! Мы получили точность на контрольном наборе 88,6 % – улучшение, демонстрирующее ценность информации о порядке слов для классификации текста. Это наша лучшая модель последовательности на данный момент! Тем не менее она все еще заметно хуже модели мешка слов.

#### 11.4.4 Когда следует использовать модели последовательности, а не модели мешка слов

Иногда говорят, что методы мешка слов устарели и что модели последовательности на основе Transformer – лучший выбор, независимо от того, какую задачу или набор данных вы рассматриваете. Это определенно не так: небольшая стопка полносвязных слоев поверх пакета биграмм остается вполне допустимым и актуальным подходом во многих случаях. На самом деле среди различных методов, которые мы опробовали на наборе данных IMDB на протяжении всей этой главы, пока что лучше всего работал пакет биграмм! Итак, в каких случаях следует предпочесть тот или иной подход?

В 2017 году мы с моей командой провели систематический анализ эффективности различных методов классификации текстов для разнообразных типов наборов текстовых данных и обнаружили удивительное эмпирическое правило, позволяющее сделать выбор между моделью мешка слов и моделью последовательности (<http://mng.bz/AOzK>), – своего рода золотую константу. Оказывается, приступая к новой задаче классификации текста, вы должны обратить особое внимание на соотношение между количеством образцов в ваших обучающих данных и средним количеством слов в образце (рис. 11.10). Если это отношение невелико – менее 1500, тогда модель мешка биграмм будет работать лучше (и в качестве бонуса она будет намного быстрее обучаться и выполнять итерации). Если это соотношение выше 1500, вам следует использовать модель последовательности. Другими словами, модели последовательностей работают лучше всего, когда доступно много обучающих данных и когда каждая выборка относительно короткая.



**Рис. 11.10** Простая эвристика для выбора модели классификации текста: отношение количества обучающих выборок к среднему количеству слов в выборке

Например, если вы классифицируете документы длиной в 1000 слов и у вас есть 100 000 документов (коэффициент 100), вам следует использовать модель биграмм. Если вы классифицируете твиты, которые в среднем состоят из 40 слов, и у вас их 50 000 (ко-

эффицент 1250), вам также следует использовать модель биграмм. Но если вы увеличите размер набора данных до 500 000 твитов (коэффициент 12 500), следует использовать кодировщик Transformer. А как насчет задачи классификации рецензий фильмов на IMDb? У нас было 20 000 обучающих образцов и среднее количество слов 233, поэтому наше эмпирическое правило указывает на модель биграмм, что совпадает с практическими результатами.

Смысл этого правила интуитивно понятен: входные данные модели последовательности представляют собой более богатое и сложное пространство, и, следовательно, требуется больше данных для отображения этого пространства. В свою очередь, небольшой набор терминов – это настолько простое пространство, что вы можете обучить логистическую регрессию, используя всего несколько сотен или тысяч образцов. Кроме того, чем короче образец, тем меньше модель может позволить себе отбрасывать какую-либо содержащуюся в нем информацию – в частности, более важным становится порядок слов, и его отбрасывание может создать двусмысленность. Предложения «этот фильм просто бомба» и «меня бомбит от этого фильма» имеют очень близкие представления униграмм, что может запутать модель мешка слов, но модель последовательности может сказать, какое из них несет отрицательный, а какое – положительный смысл. При более длинных образцах статистика по словам станет более надежной, а тема или тональность будет более очевидной только по гистограмме частотности слов.

Только имейте в виду, что это эвристическое правило было разработано специально для задачи классификации текстов. Оно не обязательно применимо для других задач NLP – например, когда дело доходит до машинного перевода, Transformer особенно хорош для очень длинных последовательностей по сравнению с RNN. Наше наблюдение является просто эмпирическим правилом, а не научным законом, поэтому можно ожидать, что оно справедливо в большинстве случаев, но не всегда.

## 11.5 Помимо классификации текста: обучение преобразованию последовательностей

Теперь вы располагаете всеми инструментами, необходимыми для решения большинства задач обработки естественного языка. Однако вы видели эти инструменты в действии только для одной задачи: классификации текста. Это чрезвычайно популярный вариант использования, но NLP – это гораздо больше, чем просто классификация. В этом разделе пойдет речь о *моделях преобразования последовательностей* (sequence-to-sequence model).

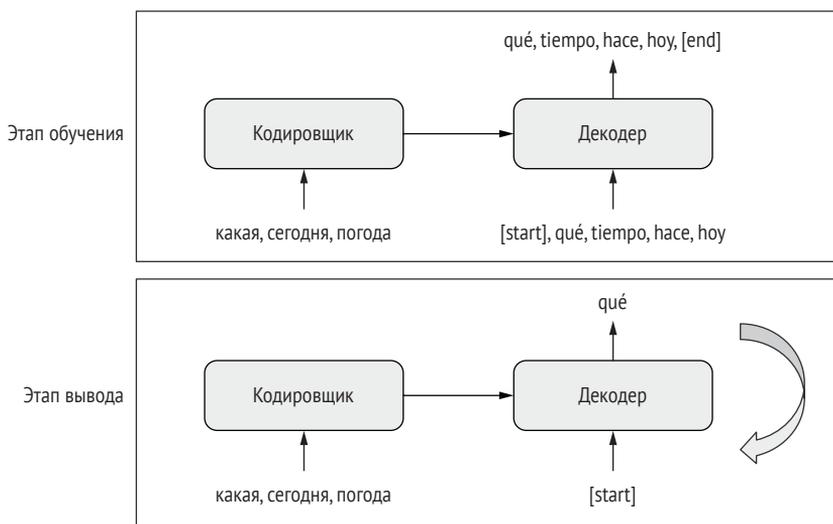
Модель преобразования последовательностей принимает в качестве входных данных последовательность (часто предложение

или абзац) и переводит ее в другую последовательность. Эта задача лежит в основе многих наиболее успешных применений NLP:

- *машинный перевод* – преобразование абзаца на исходном языке в его эквивалент на целевом языке;
- *обобщение текста* – преобразование длинного документа в более короткую версию, сохраняющую наиболее важную информацию;
- *ответ на вопрос* – преобразование входного вопроса в ответ на него;
- *чат-боты* – преобразование обращения в ответ на это обращение или преобразование истории беседы в следующий ответ в беседе;
- *генерация текста* – преобразование текстовой подсказки в абзац, завершающий подсказку;
- ... и т. д.

Общий шаблон, лежащий в основе моделей преобразования последовательностей, показан на рис. 11.11. Во время обучения:

- *кодировщик* превращает исходную последовательность в промежуточное представление;
- *декодер* обучен предсказывать следующую лексему  $i$  в целевой последовательности, просматривая как предыдущие лексемы (от 1 до  $i - 1$ ), так и закодированную исходную последовательность.



**Рис. 11.11** Обучение модели преобразования последовательностей. Исходная последовательность обрабатывается кодировщиком и затем отправляется в декодер. Декодер просматривает целевую последовательность на данный момент и прогнозирует смещение целевой последовательности на один шаг в будущем. Во время вывода мы генерируем по одному целевому токenu и возвращаем его обратно в декодер

Во время вывода у нас нет доступа к целевой последовательности – мы пытаемся предсказать ее с нуля. Поэтому нам придется генерировать ее по одному токену за раз:

- 1 получаем закодированную исходную последовательность на выходе кодировщика;
- 2 декодер начинает с просмотра закодированной исходной последовательности, а также начального токена (например, строки "[start]") и использует их для предсказания первого реального токена в выходной последовательности;
- 3 спрогнозированная на данный момент последовательность возвращается обратно в декодер, который генерирует следующий токен и т. д., пока не сгенерирует токен остановки (например, строку "[end]").

Вы можете использовать все ранее полученные знания для создания модели нового типа. Давайте посмотрим, как это можно сделать.

### 11.5.1 Пример машинного перевода

Мы продемонстрируем модель преобразования последовательности на задаче машинного перевода. Машинный перевод – это именно то, для чего была разработана архитектура Transformer! Мы начнем с рекуррентной модели последовательности и перейдем к полной архитектуре Transformer. Мы будем работать с набором данных для перевода с английского языка на испанский, доступным по адресу <http://www.manythings.org/anki/>. Начнем с загрузки набора данных:

```
download.file(
  "http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip",
  destfile = "spa-eng.zip")
zip::unzip("spa-eng.zip")
```

Текстовый файл содержит по одному примеру на строку: предложение на английском языке, за которым следует символ табуляции, а затем соответствующее предложение на испанском языке. Воспользуемся методом `readr::read_tsv()`, так как у нас есть значения, разделенные табуляцией:

```
text_file <- "spa-eng/spa.txt"
text_pairs <- text_file %>%
  readr::read_tsv(col_names = c("english", "spanish"),
                 col_types = c("cc")) %>%
  within(spanish %<>% paste("[start]", ., "[end]"))
```

Каждая строка содержит английскую фразу и ее испанский перевод, разделенные табуляцией

Читаем файл, используя `read_tsv()` (значения, разделенные табуляцией)

Двухсимвольные столбцы

Мы начинаем испанское предложение с токена «[start]» и заканчиваем токеном «[end]», чтобы соответствовать шаблону с рис. 11.11

Наши пары `text_pairs` выглядят так:

```
str(text_pairs[sample(nrow(text_pairs), 1), ])
tibble [1 × 2] (S3: tbl_df/tbl/data.frame)
 $ english: chr "I'm staying in Italy."
 $ spanish: chr "[start] Me estoy quedando en Italia. [end]"
```

Перемешаем их и разделим на обучающий, проверочный и контрольный наборы:

```
num_test_samples <- num_val_samples <-
  round(0.15 * nrow(text_pairs))
num_train_samples <- nrow(text_pairs) - num_val_samples - num_test_samples

pair_group <- sample(c(
  rep("train", num_train_samples),
  rep("test", num_test_samples),
  rep("val", num_val_samples)
))

train_pairs <- text_pairs[pair_group == "train", ]
test_pairs <- text_pairs[pair_group == "test", ]
val_pairs <- text_pairs[pair_group == "val", ]
```

Далее подготовим два отдельных слоя `TextVectorization`: один для английского и один для испанского языка. Для этого нужно настроить предварительную обработку строк:

- нам нужно сохранить токены "[start]" и "[end]", которые мы вставили. По умолчанию символы квадратных скобок [ и ] удаляются, но нам нужно сохранить их, чтобы различать слово «start» и начальный токен "[start]";
- в разных языках знаки пунктуации могут различаться! В слое `TextVectorization` для испанского языка, если мы собираемся удалить знаки препинания, нам также нужно удалить символ ¿.

Учтите, что в настоящей модели перевода знаки препинания рассматривают как отдельные токены, а не удаляют их, потому что необходимо иметь возможность генерировать предложения с правильной пунктуацией. В нашем примере для простоты мы избавимся от всех знаков препинания.

Итак, подготовим пользовательскую функцию стандартизации строк для слоя `TextVectorization` испанского языка: она сохраняет [ и ], но удаляет ¿, ¡ и все остальные символы из класса `[:punct:]`. (Двойное отрицание класса `[:punct:]` игнорируется, как если бы отрицания вообще не было. Однако наличие *внешней* группы регулярных выражений с отрицанием позволяет нам намеренно исключить символы [ и ] из класса регулярных выражений `[:punct:]`. Символ вертикальной черты | здесь применяется для добавления других специальных символов, не входящих в класс символов `[:punct:]`, например ; и ¿.)

### Листинг 11.26 Векторизация текстовых пар на английском и испанском языках

По сути, это `[[:punct:]]`, за исключением того, что в нем опущены «`[`» и «`]`» и добавлены «`¿`» и «`¡`»

```
punctuation_regex <- "[^[:^punct:][\\]]|[¿;¡]"
```

```
library(tensorflow)
```

```
custom_standardization <- function(input_string) {
```

```
  input_string %>%
```

```
  tf$strings$lower() %>%
```

```
  tf$strings$regex_replace(punctuation_regex, "")
```

```
}
```

```
input_string <- as_tensor("[start] ;corre! [end]")
```

```
custom_standardization(input_string)
```

```
tf.Tensor(b'[start] corre [end]', shape=(), dtype=string)
```

Примечание: на этот раз мы используем тензорные операции. Это позволяет отслеживать функцию в графе TensorFlow

Сохранены `[` и `]` из `[начало]` и `[конец]` и удалены `;` и `!`

**ПРЕДУПРЕЖДЕНИЕ** Регулярные выражения TensorFlow имеют незначительные отличия от таковых в R. Ознакомьтесь с официальной документацией, если собираетесь использовать расширенные регулярные выражения: <https://github.com/google/re2/wiki/Syntax>.

```
vocab_size <- 15000
```

```
sequence_length <- 20
```

Для простоты мы рассмотрим только 15 000 лучших слов на каждом языке и ограничим предложения до 20 слов

```
source_vectorization <- layer_text_vectorization(
  max_tokens = vocab_size,
  output_mode = "int",
  output_sequence_length = sequence_length
)
```

```
target_vectorization <- layer_text_vectorization(
  max_tokens = vocab_size,
  output_mode = "int",
  output_sequence_length = sequence_length + 1,
  standardize = custom_standardization
)
```

Изучение словаря каждого языка

```
adapt(source_vectorization, train_pairs$english)
```

```
adapt(target_vectorization, train_pairs$spanish)
```

Генерация испанских предложений с одним дополнительным токеном, потому что нам нужно сместить предложение на один шаг во время обучения

Теперь мы можем превратить наши данные в конвейер набора данных TensorFlow. Мы хотим, чтобы он возвращал пару (`inputs`, `target`), где `inputs` – это именованный список с двумя элементами: английское предложение `english` (ввод кодировщика) и испанское предложение `spanish` (ввод декодера), а `target` – это испанское предложение, смещенное на один шаг вперед.

**Листинг 11.27 Подготовка наборов данных для задачи перевода**

```

format_pair <- function(pair) {
  eng <- source_vectorization(pair$english)
  spa <- target_vectorization(pair$spanish)

  inputs <- list(english = eng,
                spanish = spa[NA:-2])
  targets <- spa[2:NA]
  list(inputs, targets)
}

batch_size <- 64

library(tfdatasets)
make_dataset <- function(pairs) {
  tensor_slices_dataset(pairs) %>%
  dataset_map(format_pair, num_parallel_calls = 4) %>%
  dataset_cache() %>%
  dataset_shuffle(2048) %>%
  dataset_batch(batch_size) %>%
  dataset_prefetch(16)
}

train_ds <- make_dataset(train_pairs)
val_ds <- make_dataset(val_pairs)
    
```

Слой векторизации можно вызывать как с пакетными, так и с непакетными данными. Здесь мы применяем векторизацию перед пакетной обработкой данных

Опускаем последний token в испанском предложении, чтобы входные данные и цели имели одинаковую длину. [NA:-2] удаляет последний элемент тензора

[2:NA] удаляет первый элемент тензора

Целевое испанское предложение на один шаг впереди. Оба имеют одинаковую длину (20 слов)

Используем кеширование в памяти для ускорения предварительной обработки

Так выглядят готовые данные нашего обучающего набора:

```

c(inputs, targets) %<-% iter_next(as_iterator(train_ds))
str(inputs)

List of 2
 $ english:<tf.Tensor: shape=(64, 20), dtype=int64, numpy=...>
 $ spanish:<tf.Tensor: shape=(64, 20), dtype=int64, numpy=...>

str(targets)

<tf.Tensor: shape=(64, 20), dtype=int64, numpy=...>
    
```

Итак, данные готовы – пора построить несколько моделей. Мы начнем с рекуррентной модели последовательностей, а затем перейдем к модели с архитектурой Transformer.

### 11.5.3 Рекуррентная модель преобразования последовательностей

Рекуррентные нейронные сети доминировали в области преобразования последовательностей с 2015 по 2017 год, прежде чем их обошел Transformer. Они служили основой для многих реальных систем

машинного перевода, как я уже упоминал в главе 10. Сервис Google Translate примерно в 2017 году работал на стеке из семи больших слоев LSTM. Но этот подход и сегодня заслуживает внимания, потому что он обеспечивает простую отправную точку для понимания моделей преобразования последовательностей.

Простейший способ использования RNN для превращения одной последовательности в другую состоит в том, чтобы сохранять выходные данные RNN на каждом временном шаге. В реализации на платформе Keras это будет выглядеть так:

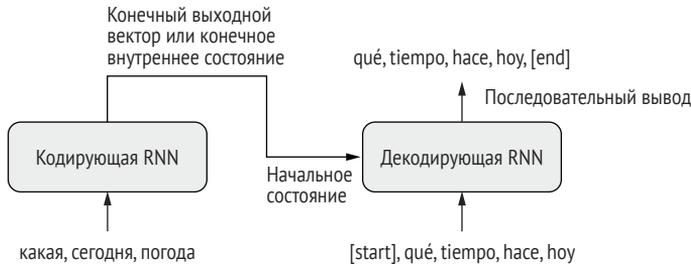
```
inputs <- layer_input(shape = c(sequence_length), dtype = "int64")
outputs <- inputs %>%
  layer_embedding(input_dim = vocab_size, output_dim = 128) %>%
  layer_lstm(32, return_sequences = TRUE) %>%
  layer_dense(vocab_size, activation = "softmax")
model <- keras_model(inputs, outputs)
```

Однако у этого подхода есть две основные проблемы:

- целевая последовательность всегда должна совпадать по длине с исходной. На практике это бывает редко. Технически это не критично, потому что вы всегда можете дополнить исходную или целевую последовательность, чтобы их длины совпадали;
- из-за пошагового характера RNN модель будет рассматривать только токены  $1 \dots N$  в исходной последовательности, чтобы предсказать токен  $N$  в целевой последовательности. Это ограничение делает архитектуру непригодной для большинства задач, особенно для перевода. Попробуйте перевести английскую фразу «The weather is nice today» (Сегодня хорошая погода) на французский – должна получиться фраза «Il fait beau aujourd'hui». Вам пришлось бы как-то предсказать «Il» только по артиклю «The», «Il fait» только по «The weather» и т. д., что просто невозможно.

Человек-переводчик обычно начинает с чтения исходного предложения целиком, прежде чем приступить к его переводу. Это особенно важно, если вы имеете дело с языковыми парами, в которых значительно различается порядок слов, например английский–японский. Именно это и делают стандартные модели преобразования последовательностей.

Правильно построенная модель преобразования последовательностей (рис. 11.12) должна сначала использовать RNN (кодировщик), чтобы преобразовать всю исходную последовательность в один вектор (или набор векторов). Это может быть последний вывод RNN или, как альтернатива, векторы окончательного внутреннего состояния нейросети. Затем вы должны использовать этот вектор (или векторы) в качестве начального состояния другой RNN (декодер), которая будет просматривать элементы  $1 \dots N$  в целевой последовательности и пытаться предсказать для нее шаг  $N + 1$ .



**Рис. 11.12 RNN для преобразования последовательностей: RNN-кодировщик применяется для создания вектора, представляющего всю исходную последовательность. Затем этот вектор используется в качестве начального состояния для RNN-декодера**

Реализуем модель с такой архитектурой в Keras, используя кодировщик и декодер на основе GRU. Выбор GRU вместо LSTM немного упрощает задачу, потому что GRU имеет только один вектор состояния, а LSTM – несколько. Начнем с кодировщика.

**Листинг 11.28 Кодер на базе GRU**

```

embed_dim <- 256
latent_dim <- 1024
source <- layer_input(c(NA), dtype = "int64",
                      name = "english")
encoded_source <- source %>%
  layer_embedding(vocab_size, embed_dim,
                 mask_zero = TRUE) %>%
  bidirectional(layer_gru(units = latent_dim,
                          merge_mode = "sum")
  
```

Здесь идет английское исходное предложение. Указание имени входных данных позволяет нам использовать fit() для модели с именованным списком входных данных

Не забывайте о маскировании: это очень важно в данном случае

Наше закодированное исходное предложение – это последний вывод двунаправленной GRU

Далее добавим декодер – простой уровень GRU, который принимает в качестве начального состояния закодированное исходное предложение. Кроме того, мы добавляем слой layer\_dense(), который для каждого шага вывода создает распределение вероятностей по испанскому словарю.

**Листинг 11.29 Декодер на основе GRU и готовая модель**

```

decoder_gru <- layer_gru(units = latent_dim, return_sequences = TRUE)

past_target <- layer_input(shape = c(NA), dtype = "int64", name = "spanish")
target_next_step <- past_target %>%
  layer_embedding(vocab_size, embed_dim,
                 mask_zero = TRUE) %>%
  decoder_gru(initial_state = encoded_source) %>%
  
```

Здесь идет испанское целевое предложение

Не забывайте о маскировании

Закодированное исходное предложение служит начальным состоянием декодера GRU

```

layer_dropout(0.5) %>%
layer_dense(vocab_size, activation = "softmax")
seq2seq_rnn <-
keras_model(inputs = list(source, past_target),
            outputs = target_next_step)

```

← Прогнозирование  
следующего токена

→ Готовая модель: сопоставляем исходное предложение и целевое предложение с целевым предложением на один шаг в будущем

Во время обучения декодер принимает на вход полную целевую последовательность, но благодаря пошаговому характеру RNN он смотрит только на токены  $1 \dots N$  на входе, чтобы предсказать токен  $N$  на выходе (который соответствует следующему токenu в последовательности, так как выходные данные должны быть смещены на один шаг). Это означает, что мы используем только информацию из прошлого, чтобы предсказывать будущее, как и должны; в противном случае мы бы обманули, и наша модель не работала бы во время логического вывода.

Теперь можно приступить к обучению модели.

### Листинг 11.30 Обучение рекуррентной модели преобразования последовательностей

```

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 метрики для мониторинга качества модели будем использовать точность на проверочном наборе во время обучения. Мы получаем точность 64 % – это означает, что в среднем модель правильно предсказывает следующее слово в испанском предложении в 64 % случаев. Однако на практике точность предсказания следующего токена не является хорошей метрикой для моделей машинного перевода, 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 логического вывода вы генерируете целевое предложение с нуля и не можете полагаться на то, что ранее сгенерированные токены будут на 100 % правильными. Если вы работаете с реальной системой машинного перевода, вы, скорее всего, будете использовать для оценки своих моделей так называемые баллы *BLEU* – показатель, который рассматривает целые сгенерированные последовательности и, по-видимому, хорошо коррелирует с человеческим восприятием качества перевода.

Наконец, воспользуемся этой моделью для вывода – выберем несколько предложений из контрольного набора и проверим, как наша модель переводит их. Мы начнем с токена "[start]" и передадим его в модель декодера вместе с закодированным английским исходным

предложением. Получив прогноз следующего токена, будем повторно вводить его в декодер несколько раз, выбирая один новый целевой токен на 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

```
spa_vocab <- get_vocabulary(target_vectorization)
max_decoded_sentence_length <- 20
decode_sequence <- function(input_sentence) {
  tokenized_input_sentence <-
    source_vectorization(array(input_sentence, dim = c(1, 1)))
  decoded_sentence <- "[start]"
  for (i in seq(max_decoded_sentence_length)) {
    tokenized_target_sentence <-
      target_vectorization(array(decoded_sentence, dim = c(1, 1)))
    next_token_predictions <- seq2seq_rnn %>%
      predict(list(tokenized_input_sentence,
                  tokenized_target_sentence))
    sampled_token_index <- which.max(next_token_predictions[1, i, ])
    sampled_token <- spa_vocab[sampled_token_index]
    decoded_sentence <- paste(decoded_sentence, sampled_token)
    if (sampled_token == "[end]") break
  }
  decoded_sentence
}
for (i in seq(20)) {
  input_sentence <- sample(test_pairs$english, 1)
  print(input_sentence)
  print(decode_sequence(input_sentence))
  print("-")
}

[1] "Does this dress look OK on me?"
[1] "[start] este vestido me parece bien [UNK] [end]"
[1] "-"
...
```

Начальный токен

Подготовка словаря для конвертации предсказаний индекса токена в строковые токены

Выборка следующего токена

Условие выхода: достижение максимальной длины предложения или стоп-токена

Конвертация следующего предсказания токена в строку и добавление к сгенерированному предложению

Метод `decode_sequence()` работает хорошо, хотя, возможно, немного медленнее, чем хотелось бы. Один из простых способов ускорить подобный непосредственно выполняемый код – использовать обертку `tf_function()`, с которой познакомились в главе 7. Давайте перепишем `decode_sentence()` так, чтобы он компилировался с помощью `tf_function()`. Это означает, что вместо использования непосредственно выполняемых функций R, таких как `seq()`, `predict()` и `which.max()`, мы будем использовать эквиваленты TensorFlow, такие как `tf$range()`, прямой вызов `model()` и `tf$argmax()`.

Поскольку `tf$range()` и `tf$argmax()` возвращают значение, начинающееся с 0, мы определим локальный аргумент функции: `option(tensorflow.extract.style = "python")`. Это изменит поведение индексов для тензоров, которые также используют отсчет от 0.

```
tf_decode_sequence <- tf_function(function(input_sentence) {
  withr::local_options(
    tensorflow.extract.style = "python")
  tokenized_input_sentence <- input_sentence %>%
    as_tensor(shape = c(1, 1)) %>%
    source_vectorization()
  spa_vocab <- as_tensor(spa_vocab)
  decoded_sentence <- as_tensor("[start]", shape = c(1, 1))
  for (i in tf$range(as.integer(max_decoded_sentence_length))) {
    tokenized_target_sentence <- decoded_sentence %>%
      target_vectorization()
    next_token_predictions <-
      seq2seq_rnn(list(tokenized_input_sentence,
                      tokenized_target_sentence))
    sampled_token_index <-
      tf$argmax(next_token_predictions[0, i, ])
    sampled_token <- spa_vocab[sampled_token_index]
    decoded_sentence <-
      tf$strings$join(c(decoded_sentence, sampled_token),
                      separator = " ")
    if (sampled_token == "[end]")
      break
  }
  decoded_sentence
})
for (i in seq(20)) {
  input_sentence <- sample(test_pairs$english, 1)
  cat(input_sentence, "\n")
  cat(input_sentence %>% as_tensor() %>%
      tf_decode_sequence() %>% as.character(), "\n")
  cat("-\n")
}
```

Теперь все подмножества Тензор после [ будут с отсчетом от 0, пока эта функция существует

i из tf\$range() отсчитывается от 0

tf\$argmax() возвращает индекс, отсчитываемый от 0

Преобразование в тензор перед вызовом tf\_decode\_sequence(), затем преобразование вывода обратно в символ R

Наш декодер `tf_decode_sentence()` работает примерно в 10 раз быстрее, чем исходная версия. Неплохо!

Обратите внимание, что этот механизм вывода, хотя и очень прост, довольно неэффективен, потому что мы повторно обрабатываем все исходное предложение и все сгенерированное целевое предложение

каждый раз, когда пробуем новое слово. На практике кодировщик и декодер рассматривают как две отдельные модели, и декодер будет выполнять только один шаг на каждой итерации выборки токенов, повторно используя свое предыдущее внутреннее состояние.

В листинге 11.32 показаны результаты перевода. Наша модель не плохо работает в качестве учебного примера, хотя и допускает много ошибок.

### Листинг 11.32 Некоторые примеры результатов перевода при помощи рекуррентной модели

```
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]
```

Есть много способов улучшить эту примитивную модель: мы могли бы использовать глубокий стек рекуррентных слоев как для кодировщика, так и для декодера (хотя для декодера это немного усложняет управление состоянием). Мы могли бы использовать LSTM вместо GRU и т. д. Однако даже после всех усовершенствований применение RNN к задаче преобразования последовательностей имеет несколько фундаментальных ограничений:

- представление исходной последовательности должно полностью храниться в векторе (векторах) состояния кодировщика, что накладывает значительные ограничения на размер и сложность предложений. Это похоже на то, как если бы человек полностью переводил предложение по памяти, не заглядывая дважды в исходное предложение при переводе;
- RNN с трудом справляются с очень длинными последовательностями, потому что они склонны постепенно забывать о прошлом – к тому времени, когда вы достигаете 100-го токена в любой последовательности, в сети остается мало информации о начале последовательности. Это означает, что модели на основе RNN не могут хранить долгосрочный контекст, что может быть необходимо для перевода длинных документов.

Это основные причины, по которым сообщество машинного обучения в задачах преобразования последовательностей перешло к использованию архитектуры Transformer. Давайте разберемся, как она работает.

### 11.5.4 Преобразование последовательностей с Transformer

Обучаемое преобразование последовательностей – это задача, в которой архитектура Transformer действительно блистает успехами. Механизм нейронного внимания позволяет моделям Transformer успешно обрабатывать последовательности, которые значительно длиннее и сложнее, чем могут осилить RNN.

Если вам, допустим, нужно перевести предложение с английского на испанский, вы не станете читать английское предложение по одному слову, запоминать его значение, а затем генерировать испанское предложение по одному слову за раз. Такой подход может сработать для предложения из пяти слов, но вряд ли сработает для всего абзаца. Вместо этого вам придется перемещаться между исходным предложением и вашим переводом и обращать внимание на разные слова в исходном тексте, по мере того как вы записываете разные части перевода.

По такому же принципу работает механизм нейронного внимания на основе архитектуры Transformer. Вы уже знакомы с кодировщиком Transformer, который использует внутреннее внимание для создания контекстно-зависимых представлений каждого токена (их еще называют *лексемами*) во входной последовательности. В преобразователе последовательностей кодирующая часть сети-трансформера, естественно, будет играть роль типичного кодировщика, который считывает исходную последовательность и создает ее закодированное представление. Однако, в отличие от нашего предыдущего кодировщика RNN, кодировщик Transformer сохраняет закодированное представление в последовательном формате: это последовательность контекстно-зависимых векторов встраивания.

Вторая половина модели – *декодер Transformer*. Как и декодер RNN, он считывает токены  $1...N$  в целевой последовательности и пытается предсказать токен  $N + 1$ . Важно отметить, что 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, – очень похоже на поведение переводчика-человека. Вспомните модель «запрос–ключ–значение»: в декодере Transformer целевая последовательность служит «запросом» внимания, который используется для более пристального внимания к различным частям исходной последовательности (которая, в свою очередь, играет роли как ключей, так и значений).

#### ДЕКОДЕР В АРХИТЕКТУРЕ TRANSFORMER

На рис. 11.13 показан полный преобразователь последовательностей. Взгляните на внутреннюю структуру декодера – он очень похож на кодировщик Transformer, за исключением того, что между блоком внутреннего внимания, применяемым к целевой последо-

вательности, и полносвязными слоями блока вывода вставлен дополнительный блок внимания.

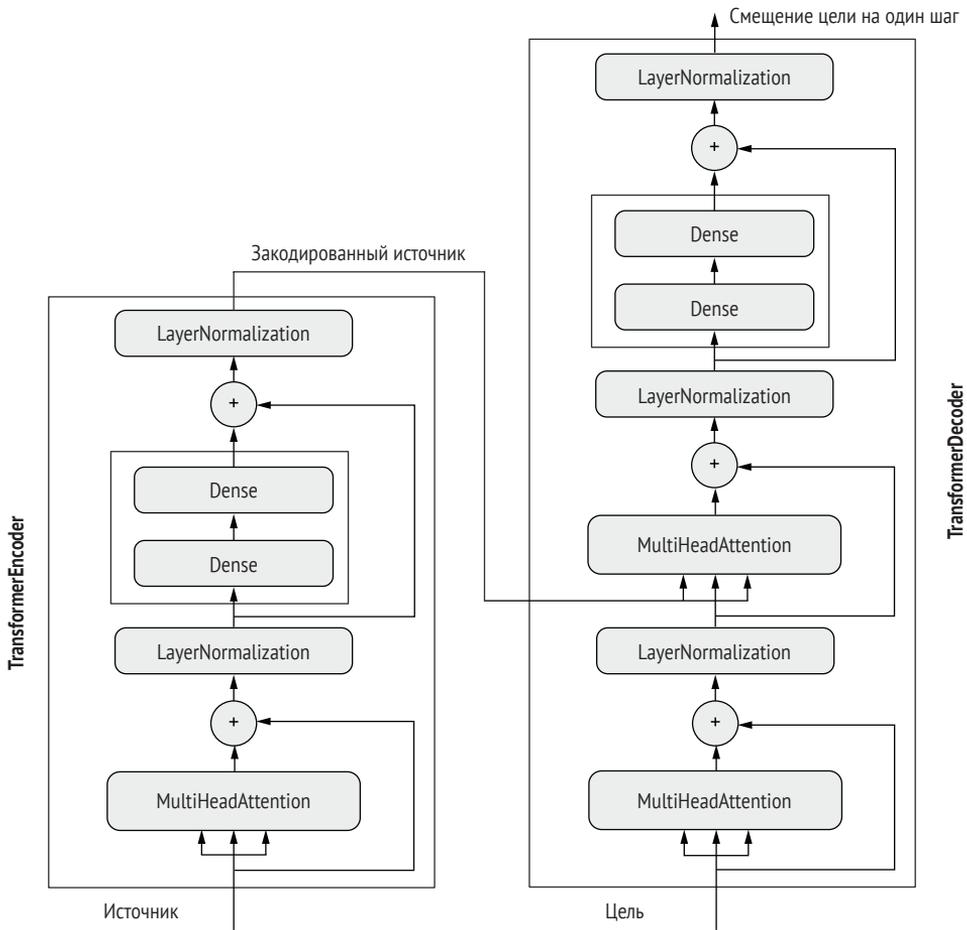


Рис. 11.13 TransformerDecoder похож на TransformerEncoder, за исключением того, что он имеет дополнительный блок внимания, где ключи и значения представляют собой исходную последовательность, закодированную в TransformerEncoder. Вместе кодировщик и декодер образуют полную архитектуру Transformer

Реализация модели показана в листинге 11.33. Как и в случае с TransformerEncoder, мы создаем новый класс слоя. Прежде чем перейти к методу call(), где происходит действие, определим конструктор класса, содержащего слои, которые нам понадобятся.

**Листинг 11.33 TransformerDecoder**

```
layer_transformer_decoder <- new_layer_class(
  classname = "TransformerDecoder",
```

```

initialize = function(embed_dim, dense_dim, num_heads, ...) {
  super$initialize(...)
  self$embed_dim <- embed_dim
  self$dense_dim <- dense_dim
  self$num_heads <- num_heads
  self$attention_1 <- layer_multi_head_attention(num_heads = num_heads,
                                                key_dim = embed_dim)
  self$attention_2 <- layer_multi_head_attention(num_heads = num_heads,
                                                key_dim = embed_dim)

  self$dense_proj <- keras_model_sequential() %>%
    layer_dense(dense_dim, activation = "relu") %>%
    layer_dense(embed_dim)

  self$layernorm_1 <- layer_layer_normalization()
  self$layernorm_2 <- layer_layer_normalization()
  self$layernorm_3 <- layer_layer_normalization()
  self$supports_masking <- TRUE
},

get_config = function() {
  config <- super$get_config()
  for (name in c("embed_dim", "num_heads", "dense_dim"))
    config[[name]] <- self[[name]]
  config
},

```

Этот атрибут гарантирует, что слой будет распространять свою входную маску на свои выходные данные; маскировка в Keras явно включена. Если вы передаете маску слою, который не реализует `compute_mask()` и не предоставляет этот атрибут `supports_masking`, это ошибка

Метод `call()` представляет собой почти буквальное воплощение диаграммы связности с рис. 11.13. Но есть еще одна деталь, которую мы должны принять во внимание: *каузальное заполнение*. Каузальное заполнение абсолютно необходимо для успешного обучения преобразователя последовательностей. В отличие от RNN, которая просматривает свои входные данные шаг за шагом и, таким образом, будет иметь доступ только к шагам  $1 \dots N$  для генерации выходного шага  $N$  (который является токеном  $N + 1$  в целевой последовательности), `TransformerDecoder` не зависит от порядка: он просматривает всю целевую последовательность сразу. Если бы ему было разрешено использовать весь ввод, он бы просто научился копировать входной шаг  $N + 1$  в ячейку  $N$  на выходе. Таким образом, модель достигла бы идеальной точности обучения, но, конечно, на практике она была бы совершенно бесполезна, потому что шаги ввода, превышающие  $N$ , недоступны.

Мы применим простое решение – замаскируем верхнюю половину парной матрицы внимания, чтобы модель не обращала внимания на информацию из будущего, т. е. при создании целевого токена  $N + 1$  следует использовать только информацию из токенов  $1 \dots N$  в целевой последовательности. Для этого мы добавим в наш `TransformerDecoder` метод `get_causal_attention_mask(inputs)`, чтобы по-

лучить маску внимания, которую мы можем передать нашим слоям MultiHeadAttention.

**Листинг 11.34** Метод TransformerDecoder, создающий каузальную маску

```

get_causal_attention_mask = function(inputs) {
  c(batch_size, sequence_length, .) %<-% ← Третья ось – encoding_length;
  tf$unstack(tf$shape(inputs))           мы не используем ее здесь

  x <- tf$range(sequence_length) ← Целочисленная последовательность
  i <- x[, tf$newaxis]             [0, 1, 2, ... sequence_length - 1]
  j <- x[tf$newaxis, ]
  mask <- tf$cast(i >= j, "int32") ← Используем изменение размерности
                                     тензора в нашей операции >=.
                                     Приводим dtype bool к int32

  tf$tile(mask[tf$newaxis, , ], ← Добавляем размер пакета в маску, а затем размещаем (rep())
    tf$stack(c(batch_size, 1L, 1L))) ← вдоль оси пакета batch_size раз. Возвращенный тензор имеет
                                     форму (batch_size, sequence_length, sequence_length)

  },

```

mask представляет собой квадратную матрицу формы (sequence\_length, sequence\_length), с 1 в нижнем треугольнике и 0 в остальных местах. Например, если sequence\_length равно 4, mask имеет вид:

```

tf.Tensor([[ [1 0 0 0]
            [1 1 0 0]
            [1 1 1 0]
            [1 1 1 1]], shape=(4, 4), dtype=int32)

```

Теперь мы можем написать полный метод call(), реализующий прямой проход декодера.

**Листинг 11.35** Прямой проход TransformerDecoder

```

call = function(inputs, encoder_outputs, mask = NULL) { ← Получаем
                                                         каузальную
                                                         маску

  causal_mask <- self$get_causal_attention_mask(inputs) ← Маска, предоставляемая в вызове, является
                                                         маской заполнения (она описывает места
                                                         заполнения в целевой последовательности)

  if (is.null(mask)) ←
    mask <- causal_mask
  else
    mask %<>% { tf$minimum(tf$cast(., tf$newaxis, ), "int32"),
              causal_mask }

  inputs %>%
    { self$attention_1(query = ., value = ., key = ., ← Передайте причинную маску
      sequence_length = sequence_length,
      attention_mask = causal_mask) + . } %>% ←
    self$layernorm_1() %>%

```

Объединяем маску заполнения с каузальной маской

Выход attention\_1() с добавленным остатком передается в layernorm\_1()

```

        Используем encoder_outputs, предоставленные в вызове,
        в качестве значения и ключа для warning_2()
        { self$attention_2(query = .,
                          value = encoder_outputs,
                          key = encoder_outputs,
                          attention_mask = mask) + . } %>%
    self$layernorm_2() %>%
    { self$dense_proj(.) + . } %>%
    self$layernorm_3()
  })
  Выход attention_2() с добавленным
  остатком передается в layernorm_2()
  Выход dense_proj() с добавленным
  остатком передается в layernorm_3()
  Передаем комбинированную
  маску второму слою внимания,
  который связывает исходную
  последовательность с целевой
  последовательностью

```

## Окончательная модель для машинного перевода

Итак, у нас есть полная модель на основе архитектуры Transformer, которую мы будем обучать. Она сопоставляет исходную и целевую последовательности с целевой последовательностью на один шаг в будущем. В ней соединяются функциональные компоненты, которые мы создали раньше: слои `PositionalEmbedding`, `TransformerEncoder` и `TransformerDecoder`. Нужно отметить, что и `TransformerEncoder`, и `TransformerDecoder` не зависят от формы, поэтому вы можете объединить несколько слоев в стек для создания более мощного кодировщика или декодера. В нашем примере для простоты используется по одному экземпляру каждого слоя.

### Листинг 11.36 Сквозная модель Transformer

```

embed_dim <- 256
dense_dim <- 2048
num_heads <- 8
Кодируем исходную
последовательность

encoder_inputs <- layer_input(shape(NA), dtype = "int64", name = "english")
encoder_outputs <- encoder_inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  layer_transformer_encoder(embed_dim, dense_dim, num_heads)
  Передаем NULL в качестве первого аргумента, чтобы
  экземпляр слоя создавался и возвращался напрямую,
  ни с чем не смешиваясь

transformer_decoder <-
  layer_transformer_decoder(NULL, embed_dim, dense_dim, num_heads)

decoder_inputs <- layer_input(shape(NA), dtype = "int64", name = "spanish")
decoder_outputs <- decoder_inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  transformer_decoder(., encoder_outputs) %>%
  layer_dropout(0.5) %>%
  layer_dense(vocab_size, activation = "softmax")
  Предсказываем слово для каждой
  выходной позиции

transformer <- keras_model(list(encoder_inputs, decoder_inputs),
                             decoder_outputs)
  Кодируем целевое предложение и объединяем его
  с закодированным исходным предложением

```

Обучив готовую модель (листинг 11.37), мы получаем точность 67 %, что заметно выше, чем у модели на основе GRU.

**Листинг 11.37 Обучение преобразователя последовательностей**

```
transformer %>%
  compile(optimizer = "rmsprop",
          loss = "sparse_categorical_crossentropy",
          metrics = "accuracy")

transformer %>%
  fit(train_ds, epochs = 30, validation_data = val_ds)
```

Теперь попробуем использовать обученную модель для перевода незнакомых ей английских предложений из тестового набора (листинг 11.38). Условия задачи в основном идентичны тем, что мы использовали для модели преобразования последовательностей на основе RNN; мы заменили seq2seq\_rnn на transformer и удалили дополнительный токен, предназначенный для добавления слоя target\_vectorization().

**Листинг 11.38 Перевод новых предложений с помощью модели Transformer**

```
tf_decode_sequence <- tf_function(function(input_sentence) {
  withr::local_options(tensorflow.extract.style = "python")

  tokenized_input_sentence <- input_sentence %>%
    as_tensor(shape = c(1, 1)) %>%
    source_vectorization()
  spa_vocab <- as_tensor(spa_vocab)
  decoded_sentence <- as_tensor("[start]", shape = c(1, 1))

  for (i in tf$range(as.integer(max_decoded_sentence_length))) {

    tokenized_target_sentence <-
      target_vectorization(decoded_sentence)[, NA:-1]

    next_token_predictions <-
      transformer(list(tokenized_input_sentence,
                      tokenized_target_sentence))

    sampled_token_index <- tf$argmax(next_token_predictions[0, i, ])
    sampled_token <- spa_vocab[sampled_token_index]
    decoded_sentence <-
      tf$strings$join(c(decoded_sentence, sampled_token),
                     separator = " ")

    if (sampled_token == "[end]")
      break
  }
}
```

Извлекаем следующий токен

Условие завершения

Отбрасываем последний токен, так как стиль Python не включает конец среза

Преобразовываем следующее предсказание токена в строку и добавляем ее к сгенерированному предложению

```

    decoded_sentence

})

for (i in sample.int(nrow(test_pairs), 20)) {
  c(input_sentence, correct_translation) %<-% test_pairs[i, ]
  cat(input_sentence, "\n")
  cat(input_sentence %>% as_tensor() %>%
      tf_decode_sequence() %>% as.character(), "\n")
  cat("-\n")
}

```

Субъективно модель на основе архитектуры Transformer работает значительно лучше, чем на основе GRU. Это все еще демонстрационная модель, но уже лучшая из демонстрационных моделей.

### Листинг 11.39 Некоторые примеры результатов работы модели перевода Transformer

```

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 creo que soy [end]
-
It may have rained a little last night.
[start] puede que llueve un poco el pasado [end]

```

Хотя исходное предложение не было гендерно ориентированным, в этом переводе предполагается, что говорящий – мужчина. Модели перевода часто делают необоснованные предположения о своих входных данных, что приводит к алгоритмической предвзятости. В худшем случае модель может извратить запомненную информацию настолько, что она не будет иметь ничего общего с данными, которые она в данный момент обрабатывает

На этом мы завершаем главу об обработке естественного языка – вы только что перешли от самых основ к полноценной модели типа Transformer, которая может переводить предложения с английского на испанский. Научить машины понимать язык – последняя суперспособность, которую вы можете добавить в свою коллекцию навыков.

## Краткие итоги главы

- Существуют два типа моделей NLP: *модели мешка слов*, которые обрабатывают наборы слов или  $N$ -грамм без учета их порядка, и *модели последовательности*, которые учитывают порядок слов.

Модель мешка слов состоит из полносвязных слоев, тогда как модель последовательности может быть рекуррентной сетью, одномерной сверточной сетью или сетью-трансформером.

- Когда речь идет о задаче классификации текста, соотношение между количеством выборок в ваших обучающих данных и средним количеством слов в выборке служит критерием предпочтения между моделью мешка слов и моделью последовательности.
- *Встраивания слов* – это векторные пространства, в которых семантические отношения между словами моделируются как отношения расстояния между векторами, представляющими эти слова.
- *Модель преобразования последовательностей* – это универсальная мощная обучаемая структура, которую можно применять для решения многих задач обработки естественного языка, включая машинный перевод. Модель преобразования последовательностей состоит из кодировщика, который обрабатывает исходную последовательность, и декодера, который пытается предсказать будущие токены в новой последовательности, просматривая прошлые токены с учетом исходной последовательности, обработанной кодировщиком.
- *Нейронное внимание* – это способ создания контекстно-зависимых представлений слов. Это основа архитектуры Transformer.
- Архитектура Transformer, состоящая из блоков TransformerEncoder и TransformerDecoder, демонстрирует отличные результаты в задачах преобразования последовательностей. Первую часть, TransformerEncoder, также можно использовать для классификации текста или любой задачи NLP с одним входом.

# 12

## Генеративные модели глубокого обучения

---

### *Эта глава охватывает следующие темы:*

- генерирование текста с помощью Keras;
- реализация алгоритма DeepDream;
- нейронный перенос 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 несколько лет от этого недоверия не осталось и следа. Летом 2015 года мы развлекались с алгоритмом Google DeepDream, превращавшим изображения в психоделическую мешанину из собачьих глаз и парейдолических артефактов; в 2016 году мы использовали приложение Prisma для превращения фотографий в картины разных стилей. Летом 2016 года вышел экспериментальный короткометражный фильм

Sunspring, снятый по сценарию, написанному алгоритмом долгой краткосрочной памяти (Long Short-Term Memory, LSTM). Возможно, вам довелось слушать музыку, сочиненную нейронной сетью.

Конечно, художественные произведения, созданные ИИ, пока изобилуют странными артефактами. Искусственный интеллект еще не может состязаться 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. Только наша человеческая интерпретация придает 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.

Яннис Ксенакис, пионер электронной и алгоритмической музыки, прекрасно выразил ту же идею в 1960-х в контексте применения технологий автоматизации к музыкальной композиции<sup>1</sup>:

*Свободный от утомительных вычислений, композитор способен посвятить себя общим проблемам, которые создает новая музыкальная форма, и исследовать самые потаенные*

---

<sup>1</sup> Iannis Xenakis, *Musiques formelles: nouveaux principes formels de composition musicale*, special issue of *La Revue musicale*, nos. 253–254 (1963).

*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 чем-то вроде пилота: он нажимает кнопки, вводит координаты и управляет космическим кораблем, плывущим в пространстве звуков, через звуковые созвездия и галактики, которые прежде он мог видеть только во снах.*

В этой главе мы исследуем потенциал глубокого обучения для расширения творческих возможностей. Мы рассмотрим приемы генерирования последовательностей данных (которые можно использовать для создания текста или музыки), алгоритм DeepDream и методы создания изображений с применением 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. Итак, начнем.

## 12.1 Генерирование текста с помощью Keras

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 генеративных сетей

В конце 2014 года мало кто был знаком с аббревиатурой LSTM, даже в сообществе машинного обучения. Успешные применения методов генерации последовательностей данных с помощью рекуррентных сетей начали приобретать широкую известность только в 2016 году. Но сами методы имеют довольно давнюю историю, начиная с раз-

работки алгоритма LSTM в 1971 году (обсуждается в главе 10). Этот новый алгоритм первое время использовался для посимвольной генерации текстов.

В 2002 году Дуглас Эк, в то время работавший в лаборатории Шмидхубера в Швейцарии, впервые применил LSTM для создания музыки и получил 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 для реализации хороших идей требуется 15 лет.

В конце 2000-х – начале 2010-х Алекс Грейвз (Alex Graves) проделал важную новаторскую работу по использованию рекуррентных сетей для генерации последовательностей данных. В частности, в 2013 году он работал над комбинацией рекуррентной и полносвязной сетей для получения человекоподобного почерка, используя временные последовательности позиций ручки, и эта работа расценивается некоторыми как поворотный момент<sup>1</sup>. Именно это конкретное применение нейронных сетей подтолкнуло меня начать разработку фреймворка Keras. В 2013 году Грейвз оставил похожее закомментированное замечание, скрытое в файле LaTeX, выгруженном на сервер препринтов arXiv: «генерация последовательностей данных – это самая близкая к воплощению мечта компьютерного мира». Несколько лет спустя мы принимаем такие разработки как нечто само собой разумеющееся, однако в то время трудно было наблюдать за демонстрациями Грейвза и не приходиться в восторг от открывающихся возможностей. В период с 2015 по 2017 год рекуррентные нейронные сети успешно применяли для генерации текста и диалогов, музыки и синтеза речи.

Затем, примерно в 2017–2018 годах, благодаря появлению архитектуры Transformer рекуррентные нейронные сети нашли применение не только в задачах обработки естественного языка, но и в генеративных моделях последовательностей, в частности в *языковых моделях* (задача генерации текста на уровне слов). Самым известным примером может служить основанная на архитектуре Transformer модель генерации текста GPT-3 со 175 млрд параметров, обученная стартапом OpenAI на огромном текстовом корпусе, включая большинство книг, доступных в цифровом виде, Википедии и большей части текстовых данных, полученных при сканировании глобального интернета. Модель GPT-3 попала в заголовки всех мировых СМИ в 2020 году из-за своей способности генерировать целые абзацы правдоподобно звучащего текста практически на любую тему – ошеломляющий эффект, напоминающий взрыв интереса к нейросетям после выдающихся успехов моделей глубокого обучения в распознавании изображений.

---

<sup>1</sup> Alex Graves, *Generating Sequences with Recurrent Neural Networks*, arXiv (2013), <https://arxiv.org/abs/1308.0850>.

### 12.1.2 Как генерируют последовательности данных?

Универсальный способ генерации последовательностей данных с применением методов глубокого обучения заключается в обучении модели (обычно это Transformer или рекуррентная сеть) для прогнозирования следующего токена или следующих нескольких токенов в последовательности, опираясь на предыдущие токены. Например, для входной последовательности «the cat is on the» сеть обучается предсказывать следующее слово «mat». 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.

Когда у вас есть такая обученная языковая модель, вы можете использовать ее для *выбора элементов* (пошагового создания новых последовательностей): вы вводите в нее начальную строку текста (называемую *обуславливающими данными*), просите ее сгенерировать следующий символ или следующее слово (вы даже можете сгенерировать несколько токенов одновременно), добавляете сгенерированный вывод обратно к входным данным и повторяете 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.

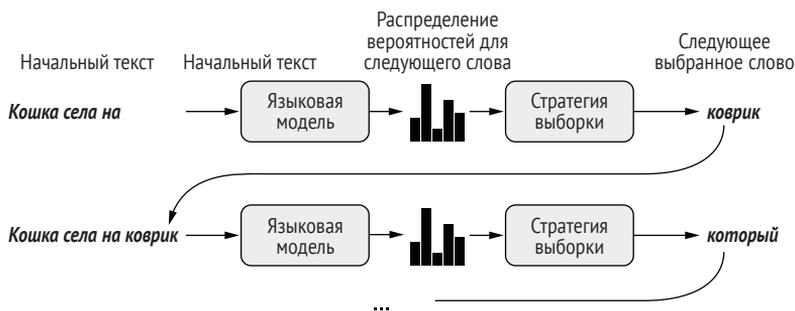


Рис. 12.1 Процесс генерации текста по словам с помощью языковой модели

### 12.1.3 Важность стратегии выбора

При генерации текста решающее значение имеет то, как вы выбираете следующий токен. Наивное решение – *жадный выбор*, когда выбирается символ с большей вероятностью. 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 может стать следующим в предложении с вероятностью 0,3, согласно модели, мы выберем его в 30 % случаев. Обратите внимание, что жадный выбор тоже можно использовать для выбора слов из распределения вероятностей, когда какое-то слово имеет вероятность 1, а все остальные – вероятность 0.

Вероятностный выбор из вектора *softmax*, возвращаемого моделью, является хорошим решением: он позволяет время от времени появляться в выводе даже маловероятным символам, генерировать более интересные предложения и иногда демонстрировать творческую жилку, придумывая новые, реалистично звучащие слова, которые отсутствуют в обучающих данных. Однако здесь есть одна проблема: эта стратегия не предусматривает возможности *управлять величиной случайности* в процессе выбора.

Зачем может понадобиться увеличивать или уменьшать случайную составляющую? Рассмотрим крайний случай: чисто случайный выбор, когда следующий символ выбирается из равномерно распределенных вероятностей и каждый символ одинаково вероятен. Эта схема имеет максимальную случайность; иными словами, это распределение вероятностей имеет максимальную энтропию. Естественно, она не произведет ничего интересного. С другой стороны, жадный выбор тоже не производит ничего интересного и не имеет случайной составляющей: соответствующее распределение вероятностей имеет минимальную энтропию. Выбор из «реального» распределения вероятностей – распределения, возвращаемого функцией *softmax*, – находится между этими двумя крайностями. Но есть еще множество других промежуточных точек с большей или меньшей энтропией, которые вы, возможно, захотите исследовать. Меньшая энтропия позволит генерировать последовательности с более предсказуемой структурой (и 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, являемся мы, люди, интересность оказывается весьма субъективной величиной, и поэтому нельзя сказать наперед, где лежит точка оптимальной энтропии.

Для управления значением 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.

Чем выше температура, тем больше энтропия распределения вероятностей и тем более неожиданными и менее структурированными будут генерируемые данные. Чем меньше температура, тем меньше будет величина случайной составляющей и тем более предсказуемыми будут генерируемые данные (см. рис. 12.2).

### Листинг 12.1 Повторное взвешивание распределения вероятностей для другой температуры

`original_distribution` – это одномерный массив значений вероятности, сумма которых должна равняться 1. `temperature` – это фактор, определяющий энтропию выходного распределения

```
reweight_distribution <-
function(original_distribution, temperature = 0.5) {
  original_distribution %>%
    { exp(log(.) / temperature) } %>%
    { . / sum(.) }
}
```

`reweight_distribution()` будет работать как для одномерных векторов R, так и для одномерных тензоров Tensorflow, потому что `exp`, `log`, `/` и `sum` встроены в R

Возвращает обновленную версию исходного распределения. Сумма распределения больше не может быть равна 1, поэтому делим ее на сумму, чтобы получить новое распределение

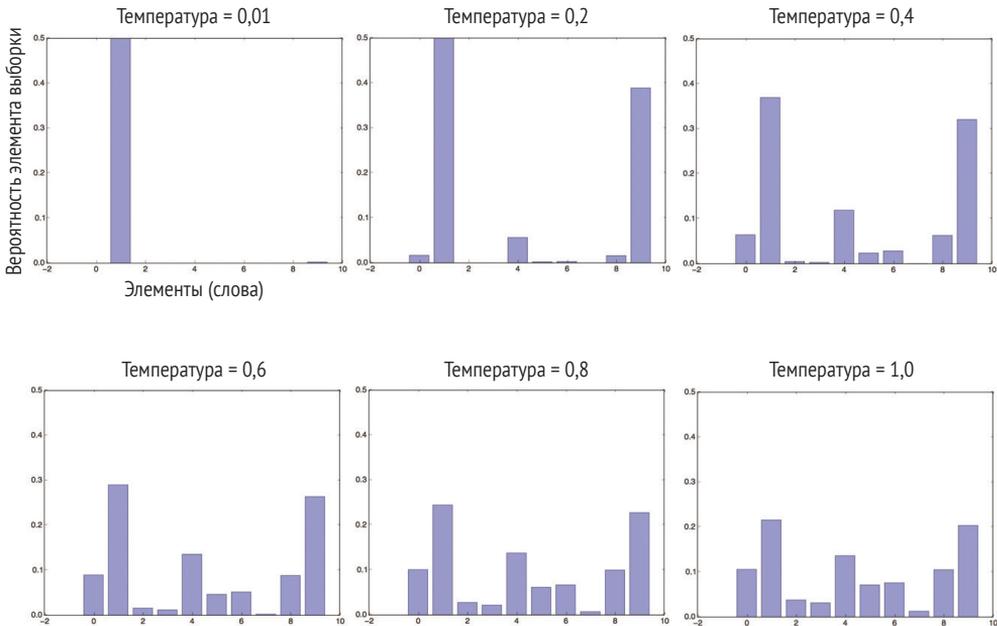


Рис. 12.2 Различные перевзвешивания одного распределения вероятностей: низкая температура = высокая предсказуемость; высокая температура = более случайный выбор

## 12.1.4 Реализация генерации текста с помощью Keras

Реализуем эти идеи на практике с помощью Keras. Первое, что нам понадобится, – это как можно больше 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 работать с набором данных обзоров фильмов IMDb из предыдущей главы и научим модель создавать обзоры фильмов, которые никогда раньше не существовали. Следовательно, у нас получится языковая модель, отражающая стиль и темы конкретных обзоров фильмов, а не общая модель английского языка.

### Подготовка данных

Как и в предыдущей главе, загрузим и распакуем набор данных обзоров фильмов IMDb. (Это тот же набор данных, с которым мы работали в главе 11.)

#### Листинг 12.2 Загрузка и распаковка набора данных обзоров фильмов IMDb

```
url <- "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"
filename <- basename(url)
options(timeout = 60 * 10) ← Тайм-аут 10 минут
download.file(url, destfile = filename)
untar(filename)
```

Вы уже знакомы со структурой данных: у нас есть каталог с именем `aclImdb`, содержащий два вложенных каталога, один для отрицательных отзывов о фильмах, а другой – для положительных. На каждый отзыв приходится отдельный текстовый файл. Далее нужно вызвать `text_dataset_from_directory()` с аргументом `label_mode = NULL`, чтобы создать объект набора данных TensorFlow, который считывает эти файлы и выдает текстовое содержимое каждого файла.

#### Листинг 12.3 Создание набора данных TensorFlow из текстовых файлов (один файл = один образец)

```
library(tensorflow)
library(tfdatasets)
library(keras)
dataset <- text_dataset_from_directory(
  directory = "aclImdb",
  label_mode = NULL,
  batch_size = 256)
dataset <- dataset %>%
dataset_map( ~ tf$strings$regex_replace(.x, "<br />", " ") ←
```

Удаление HTML-тега «<br />», который встречается во многих обзорах. Это не имело большого значения для классификации текста, но мы не хотим генерировать теги «<br />» в этом примере!

Теперь воспользуемся слоем `layer_text_vectorization()` для вычисления словаря, с которым мы будем работать. Мы будем использовать только первые `sequence_length` слов каждого отклика – `layer_text_vectorization()` обрежет остальную часть при векторизации текста.

**Листинг 12.4** Подготовка `layer_text_vectorization()`

```
sequence_length <- 100
vocab_size <- 15000
text_vectorization <- layer_text_vectorization(
  max_tokens = vocab_size,
  output_mode = "int",
  output_sequence_length = sequence_length
)
adapt(text_vectorization, dataset)
```

Мы будем рассматривать только первые 15 000 самых распространенных слов – все остальные будут рассматриваться как токен вне словарного запаса «[UNK]»

Необходимо вернуть целочисленные последовательности индексов слов

Мы будем работать с входными данными и целями длиной 100 (но поскольку мы сместим цели на 1, модель фактически увидит последовательности длиной 99)

Мы воспользуемся этим слоем для создания набора данных моделирования языка, где входные образцы представляют собой векторизованные тексты, а соответствующие цели – те же тексты, смещенные на одно слово.

**Листинг 12.5** Создание набора данных языкового моделирования

```
prepare_lm_dataset <- function(text_batch) {
  vectorized_sequences <- text_vectorization(text_batch)
  x <- vectorized_sequences[, NA:-2]
  y <- vectorized_sequences[, 2:NA]
  list(x, y)
}

lm_dataset <- dataset %>%
  dataset_map(prepare_lm_dataset, num_parallel_calls = 4)
```

Преобразование текстового пакета данных (строки) в пакет целочисленных последовательностей

Создаем входные данные, отрезав последнее слово последовательностей (отбросив последний столбец)

Создаем целевые данные, сместив последовательности на 1 (отбросив первый столбец)

## МОДЕЛЬ ПРЕОБРАЗОВАНИЯ ПОСЛЕДОВАТЕЛЬНОСТЕЙ НА ОСНОВЕ АРХИТЕКТУРЫ TRANSFORMER

Мы обучим модель прогнозировать распределение вероятностей для следующего слова в предложении с учетом предшествующих

слов. Когда модель будет обучена, мы дадим ей *начальную подсказку* (prompt), выберем следующее слово, добавим это слово обратно в подсказку и будем повторять эти шаги, пока не сгенерируем короткий абзац.

Мы могли бы воспользоваться решением для прогнозирования температуры из главы 10, обучив модель, которая принимает в качестве входных данных последовательность из  $N$  слов и просто предсказывает слово  $N + 1$ . Однако у этого подхода есть несколько серьезных недостатков в контексте генерации последовательности.

Во-первых, модель научится делать прогнозы только при наличии  $N$  слов, но было бы полезно иметь возможность работать с меньшим количеством слов, чем  $N$ . В противном случае мы будем вынуждены использовать только относительно длинные подсказки (в нашей реализации  $N = 100$  слов). В главе 10 у нас не было такой потребности.

Во-вторых, многие из наших обучающих последовательностей будут значительно перекрываться. Рассмотрим случай, когда  $N = 4$ . Текст «Полное предложение должно иметь как минимум три члена: подлежащее, сказуемое и дополнение» будет использоваться для создания следующих обучающих последовательностей:

- «Полное предложение должно иметь»;
- «предложение должно иметь как»;
- «должно иметь как минимум»;
- и т. д., до конца предложения.

Модель, которая рассматривает каждую такую последовательность как независимый образец, выполняла бы много лишней работы, повторно кодируя большое количество частичных последовательностей, которые она в основном видела раньше. В главе 10 это не стало для нас проблемой, потому что у нас изначально не было такого количества обучающих образцов и нам нужно было протестировать плотные и сверточные модели, для которых переделывать работу каждый раз – единственный вариант. Мы могли бы попытаться решить проблему избыточности, используя *шаги* для прореживания наших последовательностей – пропуская несколько слов между двумя последовательными образцами. Но это уменьшило бы количество обучающих образцов, обеспечив лишь частичное решение.

Устранить эти две проблемы нам поможет *модель преобразования последовательностей*: мы будем передавать последовательности из  $N$  слов (индексированных от 1 до  $N$ ) в нашу модель и предсказывать смещение последовательности на единицу (от 2 до  $N + 1$ ). Мы будем использовать каузальную маску, чтобы гарантировать, что для любого  $i$  модель будет использовать только слова от 1 до  $i$  для предсказания слова  $i + 1$ . Это означает, что мы одновременно обучаем модель решать  $N$  в основном перекрывающихся, но разных задач: предсказание следующих слов по последовательности  $1 \leq i \leq N$  предшествующих слов (рис. 12.3). Во время генерации, даже если вы

предложите модели только одно слово, она сможет дать вам распределение вероятностей для следующих возможных слов.

Предсказание следующего слова	the cat sat on the → <b>mat</b>
	the → <b>cat sat on the mat</b>
	the cat → <b>sat on the mat</b>
Моделирование преобразования последовательностей	the cat sat → <b>on the mat</b>
	the cat sat on → <b>the mat</b>
	the cat sat on the → <b>mat</b>

**Рис. 12.3** По сравнению с простым прогнозированием следующего слова, моделирование преобразования последовательностей одновременно оптимизируется для решения нескольких задач прогнозирования

Мы могли бы использовать аналогичное преобразование последовательности для нашей задачи прогнозирования температуры в главе 10: имея последовательность из 120 почасовых точек данных, научить модель генерировать последовательность из 120 температур, смещенных на 24 часа в будущем. Нам пришлось бы решать не только начальную задачу, но и 119 связанных с ней задач прогнозирования температуры за 24 часа, учитывая  $1 \leq i < 120$  предыдущих почасовых точек данных. Если вы попытаетесь переобучить RNN в новой конфигурации, то обнаружите, что получаете похожие, но постепенно ухудшающиеся результаты, потому что ограничение решения этих дополнительных 119 связанных задач той же самой моделью немного мешает задаче, которая нас действительно интересует.

В предыдущей главе вы узнали о базовом принципе, который можно использовать для обучения модели преобразования последовательностей: пропустить исходную последовательность через кодировщик, а затем подать как закодированную, так и целевую последовательность в декодер, который пытается предсказать исходную последовательность, смещенную на один шаг. В задаче генерации текста исходной последовательности нет: модель пытается предсказать следующие токены в целевой последовательности, учитывая предыдущие токены, а это можно сделать, используя только декодер. Благодаря каузальному дополнению декодер будет смотреть лишь на слова  $1 \dots N$ , чтобы предсказать слово  $N + 1$ .

В листинге 12.6 показана реализация такой модели. Здесь повторно применяются компоненты, созданные в главе 11: `layer_positional_embedding()` и `layer_transformer_decoder()`.

#### Листинг 12.6 Простая языковая модель на основе архитектуры Transformer

```
embed_dim <- 256
latent_dim <- 2048
num_heads <- 2
```

```

transformer_decoder <-
  layer_transformer_decoder(NULL, embed_dim, latent_dim, num_heads)

inputs <- layer_input(shape(NA), dtype = "int64")
outputs <- inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  transformer_decoder(., .) %>%
  layer_dense(vocab_size, activation = "softmax")

```

Применяем softmax ко всем возможным словарным словам, выполняя вычисления на каждом шаге выходной последовательности

```

model <-
  keras_model(inputs, outputs) %>%
  compile(loss = "sparse_categorical_crossentropy",
          optimizer = "rmsprop")

```

### 12.1.5 Обратный вызов генерации текста с выборкой при разной температуре

Далее мы сгенерируем текст при помощи обратного вызова, используя разные температуры после каждой эпохи. Это позволит увидеть, как сгенерированный текст меняется, по мере того как модель начинает сходиться, а также влияние температуры на стратегию выборки. Все наши сгенерированные тексты будут начинаться с подсказки «this movie» (этот фильм).

Начнем с определения функций для генерации предложений. Позже мы будем использовать эти функции в обратном вызове.

Вектор, который мы будем использовать для преобразования индексов слов (целых чисел) обратно в строки для последующего декодирования текста

```
vocab <- get_vocabulary(text_vectorization)
```

```

sample_next <- function(predictions, temperature = 1.0) {
  predictions %>%
    reweight_distribution(temperature) %>%
    sample.int(length(.), 1, prob = .)
}

```

Температура, используемая для отбора образцов

Реализация выборки с переменной температурой из вероятностного распределения

```

generate_sentence <-
  function(model, prompt, generate_length, temperature) {

    sentence <- prompt
    for (i in seq(generate_length)) {

      model_preds <- sentence %>%
        array(dim = c(1, 1)) %>%
        text_vectorization() %>%
        predict(model, .)

      sampled_word <- model_preds %>%
        .[1, i, ] %>%
        sample_next(temperature) %>%

```

prompt используется для начала генерации текста

Повторите, сколько слов нужно сгенерировать

Подача текущей последовательности в нашу модель

Получение прогнозов для последнего временного шага...

... и использование ее для выборки нового токена...

```

vocab[.] ← ... а затем конвертация целочисленного токена в строку
sentence <- paste(sentence, sampled_word) ← Добавление нового слова к текущей
}                                       последовательности и повтор цикла
sentence
}

```

Функции `sample_next()` и `generate_sentence()` выполняют работу по генерации предложений из модели. Они, в свою очередь, вызывают `predict()` для создания прогнозов в виде массивов R, `sample.int()` для выбора следующей лексемы и создают предложение в виде строки R с помощью `paste()`.

Поскольку мы ожидаем, что текст будет состоять из нескольких предложений, имеет смысл выполнить небольшую оптимизацию, чтобы ускорить выполнение. Мы можем значительно ускорить `generate_sentence` (примерно в 25 раз), переписав ее как `tf_function()`. Для этого нам просто нужно заменить несколько функций R эквивалентами TensorFlow. Вместо `for(i in seq())` мы можем написать `for(i in tf$range())`. Мы также можем заменить `sample.int()` на `tf$random$catagorical()`, `paste()` на `tf$strings$join()` и `predict(model, .)` на `model(.)`. В следующем коде показано, как выглядят `sample_next()` и `generate_sentence()` в виде `tf_function()`:

```

tf_sample_next <- function(predictions, temperature = 1.0) {
  predictions %>%
    reweight_distribution(temperature) %>%
    { log(.[tf$newaxis, ]) } %>%
    tf$random$catagorical(1L) %>%
    tf$reshape(shape())
}
library(tfautograph)
tf_generate_sentence <- tf_function(
  function(model, prompt, generate_length, temperature) {
    withr::local_options(tensorflow.extract.style = "python")
    vocab <- as_tensor(vocab)
    sentence <- prompt %>% as_tensor(shape = c(1, 1))
    ag_loop_vars(sentence)
    for (i in tf$range(generate_length)) {
      model_preds <- sentence %>%
        text_vectorization() %>%
        model()
      sampled_word <- model_preds %>%

```

tf\$random\$catagorical() ожидает пакет логарифмических вероятностей

tf\$random\$catagorical() возвращает скалярное целое число формы (1, 1). Приведение к форме ()

Для ag\_loop\_vars() (подробнее об этом позже)

Даем компилятору подсказку, что 'sentence' - единственная переменная, которая нам нужна после итерации

```

.[0, i, ] %>%
tf_sample_next(temperature) %>%
vocab[.]

sentence <- sampled_word %>%
  { tf$strings$join(c(sentence, .), " ") }

}

sentence %>% tf$reshape(shape())
}
)

```

Преобразуем форму с (1, 1) на ().  
Обратите внимание, что `tf$strings$join()`  
сохраняет форму предложения (1, 1)  
на протяжении всей итерации

На моем компьютере генерация предложения из 50 слов занимает примерно 2,5 секунды с прямым выполнением `generate_sentence()` и 0,1 секунды с `tf_generate_sentence()`, что дает улучшение в 25 раз! Помните, что всегда имеет смысл сначала создать прототип вашего кода, запустив его с прямым выполнением для отладки, и переходить к использованию обертки `tf_function()` только после того, как он заработает, как вам нужно.

### Циклы `for` и параметр `autograph`

Проблема с отладкой прямого выполнения функций R перед обертыванием их с помощью `tf_function(fn, autograph = TRUE)` заключается в том, что параметр по умолчанию `autograph = TRUE` дает возможности, которых нет в базовом R, например возможность перебирать тензоры в цикле `for`. Во время отладки вы по-прежнему можете вычислять такие выражения, как `for(i in tf$range())` или `for(batch in tf_dataset)`, напрямую вызывая `tfautograph::autograph()`, например:

```

library(tfautograph)
autograph({
  for(i in tf$range(3L))
    print(i)
})

tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)

или

fn <- function(x) {
  for(i in x) print(i)
}
ag_fn <- autograph(fn)
ag_fn(tf$range(3))

tf.Tensor(0.0, shape=(), dtype=float32)
tf.Tensor(1.0, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)

```

В интерактивных сеансах вы можете временно глобально разрешить `if`, `while` и `for` принимать тензоры, вызвав `tfautograph::attach_ag_mask()`.

Цикл `for()`, перебирающий тензор в `tf_function()`, создает `tf$while_loop()` и наследует все те же ограничения. Каждый тензор, отслеживаемый циклом, должен сохранять неизменными форму и тип на протяжении всей итерации.

Вызов `ag_loop_vars(sentence)` дает компилятору `tf_function()` подсказку, что единственная интересующая нас переменная после цикла `for` – это `sentence`. Она информирует компилятор о том, что другие тензоры, такие как `sampled_word`, `i` и `model_preds`, являются локальными переменными цикла и могут быть безопасно оптимизированы после завершения цикла.

Обратите внимание, что итерация по обычному объекту R, например `for(i in seq(0, 49))` в `tf_function()`, не создаст `tf$while_loop()`, а вместо этого будет обработана с помощью обычной семантики R, поэтому `tf_function()` сформирует развернутый цикл (что иногда предпочтительнее для коротких циклов с фиксированным числом итераций).

В листинге 12.7 представлен обратный вызов, в котором мы будем вызывать `tf_generate_sentence()` для генерации текста во время обучения:

### Листинг 12.7 Обратный вызов для генерации текста

```
callback_text_generator <- new_callback_class(
  classname = "TextGenerator",

  initialize = function(prompt, generate_length,
                        temperatures = 1,
                        print_freq = 1L) {
    private$prompt <- as_tensor(prompt, "string")
    private$generate_length <- as_tensor(generate_length, "int32")
    private$temperatures <- as.numeric(temperatures)
    private$print_freq <- as.integer(print_freq)
  },

  on_epoch_end = function(epoch, logs = NULL) {
    if ((epoch %% private$print_freq) != 0)
      return()

    for (temperature in private$temperatures) {
      cat("== Generating with temperature", temperature, "\n")

      sentence <- tf_generate_sentence(
        self$model,
        private$prompt,
        private$generate_length,

```

Мы будем использовать различные диапазоны температур для выборки текста, чтобы продемонстрировать влияние температуры на генерацию текста

←

Это обычный цикл R `for`, непосредственно итерируемый по вектору R

←

Обратите внимание, что мы вызываем эту функцию только с тензорами и моделью, а не с числовыми или символьными векторами R

←

Они уже были переданы Tensors в `initialize()`

←

```

        as_tensor(temperature, "float32")
    )
    cat(as.character(sentence), "\n")
  }
}
)

text_gen_callback <- callback_text_generator(
  prompt = "This movie",
  generate_length = 50,
  temperatures = c(0.2, 0.5, 0.7, 1., 1.5) ←
)

```

Набор температур, с которым мы генерируем текст

Выполним `fit()` для обучения модели (листинг 12.8).

### Листинг 12.8 Обучение языковой модели

```

model %>%
  fit(lm_dataset,
      epochs = 200,
      callbacks = list(text_gen_callback))

```

Ниже показаны примеры текстов на английском языке, сгенерированные после 200 эпох обучения (Для удобства оценки качества генерации под ними дан дословный перевод на русский язык. – Прим. перев.). Знаки препинания не являются частью словаря, поэтому ни один из сгенерированных текстов не содержит знаков препинания:

- при `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* (этот фильм [UNK] оригинального фильма, и первые полчаса фильма довольно хороши но это очень хороший фильм это хороший фильм для того времени);
  - *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* (этот фильм является [UNK] фильма это фильм который настолько плох что это [UNK] фильм это фильм который настолько плох что заставляет вас смеяться и плакать одновременно это не фильм который ранее невиданный для меня);
- при `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* (этот фильм [UNK] из лучших жанровых фильмов всех времен и это не хороший фильм единственная хорошая вещь в этом фильме что я видел его впервые и я до сих пор помню что это был [UNK] фильм который я видел много лет);
  - *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* (этот фильм пустая трата времени и денег я должен сказать что этот фильм был пустой тратой времени я был удивлен увидев что фильм сделан из хорошего фильма а фильм не очень хороший но это был пустая трата времени и);

- при  $temperature=0.7$ :
  - *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* (этот фильм интересно смотреть и действительно забавно наблюдать что все персонажи очень веселые а еще кот немного похож на [UNK] [UNK] и шляпу [UNK] правила фильма можно рассказать на другой сцене спасает его от того чтобы быть позади);
  - *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* (этот фильм о [UNK] и паре молодых людей на маленькой лодке посреди ниоткуда где они могут столкнуться со стоматологом [UNK] которым они убиты [UNK] я был большим поклонником книги а оригинала я не видел так что);
- при  $temperature=1.0$ :
  - *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]* (этот фильм был интересным я чувствовал что сюжетная линия была громкой и трогательной но в целом просмотр резко контрастирует с художественным оформлением оригинала мы смотрели оригинальную версию англии однако в то время как арка была немного слишком обычной [UNK] были нынешним родителем [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* (этот фильм был шедевром далеким от сюжетной линии но этот фильм был просто захватывающим и разочаровывающим он действительно развлекает таких друзей как этот актеры в этом фильме пытаются действовать прямо от субтитров это образ и они делают из него действительно хорошее телешоу);
- при  $temperature=1.5$ :
  - *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* (этот фильм был возможно худшим фильмом о тех 80 женщинах это были странные пронциательные актеры такие как фильмы о баркерах но в великих приятелях да нет украшенного щита даже [UNK] наземный динозавр ральф должен был сыграть случившееся падает после неправильного использования [UNK] не совсем не рестливания серьезно сэм не существовал);
- *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* (этот фильм может быть настолько невероятно что сам лукас приносит нашей стране дико забавные вещи которые он имеет для ярких серьезных и сильных выступлений колин пишет более подробно до этого доминировал и это изображения шестерен сжигающих тарелку патриотизм которого мы ожидали сделать самостоятельно и другое).

Как видите, при низком значении температуры генерируются довольно скучные и повторяющиеся тексты, а иногда может произойти заикливание процесса генерации. При более высоких температурах сгенерированный текст становится более интересным, неожиданным и даже творческим. При очень высокой температуре локальная структура текста начинает разрушаться, и результат выглядит преимущественно случайным. В данном случае оптимальной температурой генерации будет значение около 0,7. Всегда экспериментируйте с несколькими стратегиями выбора! Разумный баланс между изученной структурой и случайностью – вот что делает генерацию интересной.

Обратите внимание, что, обучая модель большего размера дольше и на большем объеме данных, можно добиться генерации текста, который выглядит еще реалистичнее. Результаты работы такой модели, как GPT-3, служат хорошим примером того, что можно сделать с помощью языковых моделей (GPT-3 фактически очень близка к модели, которую мы обучали в этом примере, но с большим набором декодеров Transformer и намного большим учебным корпусом). Однако не думайте, что когда-нибудь вам удастся сгенерировать осмысленный текст, разве только по чистой случайности: вы всего лишь выбираете образцы данных из статистической модели, в которой одни слова следуют за другими. Языковые модели – это инструмент формы, а не содержания. Естественный язык – это намного больше, чем просто порядок слов. Это канал общения, способ воздействовать на мир, социальные связи, способ хранить и извлекать собственные мысли и донести их до окружающих. Язык берет свое начало в способах использования. «Языковая модель» глубокого обучения, несмотря на свое название, не отражает ни один из

этих фундаментальных аспектов языка. Модель не может общаться (ей не о чем и не с кем общаться), она не может воздействовать на мир (у нее нет агентов и намерений), она не может социализоваться, и у нее нет своих мыслей, которыми можно было бы поделиться с помощью слов. Язык – это операционная система разума, поэтому только разум может наделить язык смыслом.

Языковая модель всего лишь фиксирует и воплощает статистическую структуру изученных артефактов – книг, онлайн-обзоров фильмов, твитов, – которые мы генерируем, когда используем язык в своей жизни. Тот факт, что эти артефакты вообще имеют статистическую структуру, является побочным эффектом человеческой реализации языка. Чтобы осознать это различие, проведите мысленный эксперимент: представьте, что человеческий язык позволял бы сжимать информацию при общении почти так же, как это делают компьютеры с цифровой информацией. Язык не потерял бы своей осмысленности, но утратил статистическую структуру, что сделало бы невозможным обучение языковой модели, как мы только что это сделали.

### 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 предлагаемой моделью вероятностью и случайностью.
- Обеспечить такой баланс можно введением понятия *температуры softmax*; всегда пробуйте разные температуры, чтобы найти наиболее подходящую.

## 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 психоделическим картинам (пример на рис. 12.4), напол-

---

<sup>1</sup> Alexander Mordvintsev, Christopher Olah, and Mike Tyka, *DeepDream: A Code Example for Visualizing Neural Networks*, Google Research Blog, July 1, 2015, <http://mng.bz/xXIM>.

ненным 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.

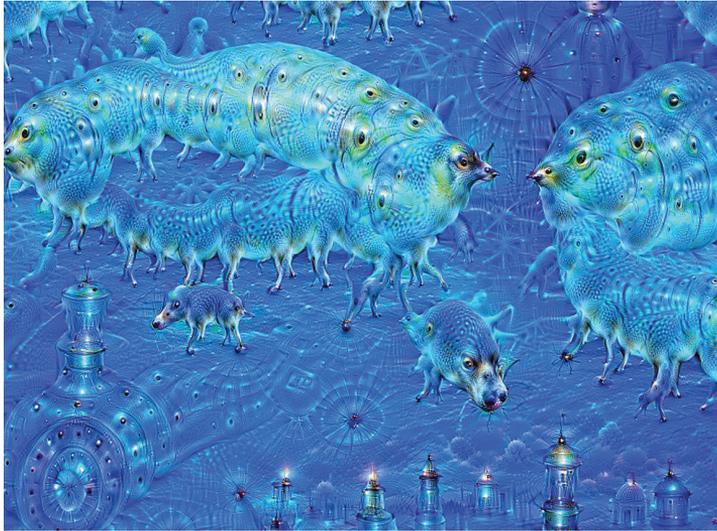


Рис. 12.4 Пример выходного изображения DeepDream

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 на более высоком слое сверточной сети. DeepDream использует ту же идею с небольшими отличиями:

- алгоритм DeepDream пытается максимизировать активацию всех слоев, а не только определенного фильтра, тем самым смешивая визуализации большего количества признаков;
- 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

Начнем с получения тестового изображения для DeepDream. Мы будем использовать вид на суровое побережье Северной Калифорнии зимой (рис. 12.5).

**Листинг 12.9** Получение тестового изображения

```
base_image_path <- get_file(  
  "coast.jpg", origin = "https://img-datasets.s3.amazonaws.com/coast.jpg")  
  
plot(as.raster(jpeg::readJPEG(base_image_path)))
```



Рис. 12.5 Наше тестовое изображение

Далее нам понадобится предварительно обученная сеть. В Keras доступны различные сверточные сети – VGG16, VGG19, Xception, ResNet50 и т. д. – все с весами, предварительно обученными на наборе ImageNet. Вы можете реализовать алгоритм DeepDream с любой из них, но выбранная вами базовая модель естественным образом повлияет на визуализацию, поскольку от выбора архитектуры зависит, какие признаки будут изучены. В исходной реализации DeepDream разработчики использовали модель Inception, и на практике известно, что Inception создает красиво выглядящие изображения DeepDreams, поэтому мы воспользуемся моделью Inception V3, которая поставляется с Keras.

**Листинг 12.10** Создание экземпляра предварительно обученной модели InceptionV3

```
model <- application_inception_v3(weights = "imagenet", include_top = FALSE)
```

Мы используем предварительно обученную сеть для создания модели извлечения признаков, которая возвращает активации нескольких промежуточных слоев, перечисленных в листинге 12.11. Для каждого слоя мы выбираем скалярный коэффициент, взвешивающий вклад слоя в потери, которые мы стремимся максимизиро-

вать в процессе градиентного восхождения. Если нужно вывести на печать полный список имен слоев, доступных вам при построении модели, воспользуйтесь командой `print(model)`.

### Листинг 12.11 Настройка вклада каждого слоя в потери DeepDream

```
layer_settings <- c(
  mixed4 = 1,
  mixed5 = 1.5,
  mixed6 = 2,
  mixed7 = 2.5
)
outputs <- list()
for(layer_name in names(layer_settings))
  outputs[[layer_name]] <-
    get_layer(model, layer_name)$output
feature_extractor <- keras_model(inputs = model$inputs,
                                outputs = outputs)
```

Слои, для которых мы стараемся максимально активировать, а также их вес в общей сумме потерь. Вы можете настроить эти параметры для получения новых визуальных эффектов

Собираем в именованный список выходные символьные тензоры из каждого слоя

Модель, которая возвращает значения активации для каждого целевого слоя (в виде именованного списка)

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 Вычисление потерь DeepDream для максимизации

```
compute_loss <- function(input_image) {
  features <- feature_extractor(input_image)

  feature_losses <- names(features) %>%
    lapply(function(name) {
      coeff <- layer_settings[[name]]
      activation <- features[[name]]
      coeff * mean(activation[, 3:-3, 3:-3, ] ^ 2)
    })
  Reduce('+', feature_losses)
}
```

Извлечение активаций

Мы избегаем краевых артефактов, исключая краевые пиксели из расчета потерь

feature\_losses представляет собой список скалярных тензоров. Суммируем потери по каждому признаку

Теперь можно настроить процесс градиентного восхождения, который мы будем выполнять для каждого масштаба (называемого октавой). На самом деле он ничем не отличается от визуализации фильтров в главе 9! Алгоритм DeepDream – это просто способ визуализации фильтра с применением кратного масштабирования.

### Листинг 12.13 Процесс градиентного восхождения DeepDream

```

gradient_ascent_step <- tf_function(
  function(image, learning_rate) {
    with(tf$GradientTape() %as% tape, {
      tape$watch(image)
      loss <- compute_loss(image)
    })
    grads <- tape$gradient(loss, image) %>%
      tf$math$l2_normalize()
    image %<>% `+`(learning_rate * grads)
    list(loss, image)
  })

gradient_ascent_loop <-
  function(image, iterations, learning_rate, max_loss = -Inf) {
    learning_rate %<>% as_tensor()
    for(i in seq(iterations)) {
      c(loss, image) %<-% gradient_ascent_step(image, learning_rate)
      loss %<>% as.numeric()
      if(loss > max_loss) break
      writeLines(sprintf(
        "... Loss value at step %i: %.2f", i, loss))
    }
    image
  }

```

Мы ускоряем шаг обучения, скомпилировав его как `tf_function()`

Вычисление градиентов потерь DeepDream по отношению к текущему изображению

Нормализация градиентов (аналогичный прием мы использовали в главе 9)

Неоднократно обновляем изображение таким образом, чтобы увеличить потери DeepDream

Запуск градиентного восхождения для заданного масштаба изображения (октавы)

Это обычный непосредственный цикл R for

Прервать цикл, если потери превысят определенный порог (чрезмерная оптимизация приведет к нежелательным искажениям изображения)

Наконец, фактический алгоритм DeepDream. Сначала определим список масштабов (также называемых октавами), в которых будут обрабатываться изображения. Мы будем обрабатывать наше изображение в трех разных октавах. Для каждой октавы мы будем выполнять цикл из 20 шагов градиентного восхождения с помощью `gradient_ascent_loop()`, чтобы максимизировать значение функции потерь, определенной ранее. После каждого цикла восхождения мы будем увеличивать изображение на 40 % ( $1,4\times$ ): начнем с обработки

небольшого изображения, а затем будем постепенно его увеличивать (рис. 12.6).

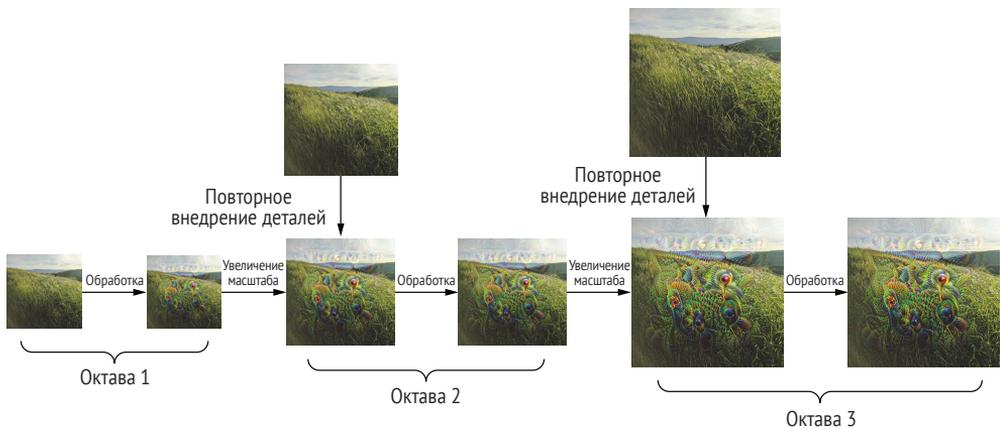


Рис. 12.6 Процесс DeepDream: последовательные масштабы пространственной обработки (октавы) и повторное внедрение деталей при увеличении масштаба

Параметры этого процесса определяются в следующем коде. Настройка этих параметров позволит вам добиться новых эффектов!

```

Шаг градиентного восхождения
└─ step <- 20
   num_octaves <- 3
   octave_scale <- 1.4
   iterations <- 30
Останов процесса градиентного восхождения
для масштаба, если потеря превысит это значение
└─ max_loss <- 15
Количество масштабов, на которых можно
запустить градиентное восхождение
Соотношение размеров между
последовательными масштабами
Количество ступеней градиентного
восхождения на масштаб

```

Нам также понадобится пара служебных функций для загрузки и сохранения изображений.

#### Листинг 12.14 Служебные функции обработки изображений

```

Служебная функция для загрузки, изменения размера
и форматирования изображений в соответствующие массивы
preprocess_image <- tf_function(function(image_path) {
  image_path %>%
  tf$io$read_file() %>%
  tf$io$decode_image() %>%
  tf$expand_dims(axis = 0L) %>%
  tf$cast("float32") %>%
  inception_v3_preprocess_input()
})
Добавление оси пакета, эквивалент
.[tf$newaxis, all_dims()], отсчет осей
идет от 0
Служебная функция для преобразования
тензорного массива в допустимое
изображение и отмены предварительной обработки
deprocess_image <- tf_function(function(img) {

```

```

saturate_case() обрезает значения к диапазону dtype [0, 255]
                                Отбрасываем первое измерение – ось пакетов (должна быть
                                размерность 1), обратная функция относительно tf$expand_dims()
    img %>%
      tf$squeeze(axis = 0L) %>%
      { (. * 127.5) + 127.5 } %>%
      tf$saturate_cast("uint8")
  })

display_image_tensor <- function(x, ..., max = 255,
                                plot_margins = c(0, 0, 0, 0)) {

  if (!is.null(plot_margins))
    withr::local_par(mar = plot_margins)

  x %>%
    as.array() %>%
    drop() %>%
    as.raster(max = max) %>%
    plot(..., interpolate = FALSE)
}

```

Изменяем масштаб так, чтобы значения из диапазона [-1, 1] были перенесены в диапазон [0, 255]

По умолчанию построение изображений без полей

Преобразование тензоров в массивы R

Преобразование в растровый формат R

### withr::local\_\*

В данном примере мы используем `withr::local_par()` для настройки `par()` перед вызовом `plot()`. Функция `local_par()` действует так же, как и `par()`, за исключением того, что она восстанавливает предыдущие настройки `par()` при выходе из функции. Использование таких функций, как `local_par()` или `local_options()`, позволяет гарантировать, что написанные вами функции не изменят глобальное состояние навсегда, что делает их более предсказуемыми и пригодными для использования в большем количестве контекстов.

Вы можете заменить `local_par()` и сделать то же самое с отдельным вызовом `on.exit()` следующим образом:

```

display_image_tensor <- function()
  <...>
  opar <- par(mar = plot_margins)
  on.exit(par(opar))
  <...>
}

```

Это внешний цикл. Чтобы избежать потери деталей изображения после каждого увеличения масштаба (в результате чего появляются эффекты размытия или мозаичности), можно использовать простой прием: после каждого изменения масштаба повторно внедрять в изображение потерянные детали, что возможно благодаря знанию, как должно выглядеть исходное изображение в увеличенном масштабе. Имея маленькое изображение с размером  $S$  и большое – с размером  $L$ , можно вычислить разницу между оригинальным изображением

жением 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 Запуск градиентного восхождения через несколько последовательных октав

```

original_img <- preprocess_image(base_image_path)
original_HxW <- dim(original_img)[2:3]

calc_octave_HxW <- function(octave) {
  as.integer(round(original_HxW / (octave_scale ^ octave)))
}

octaves <- seq(num_octaves - 1, 0) %>%
  { zip_lists(num = .,
             HxW = lapply(., calc_octave_HxW)) }

str(octaves)

List of 3
 $ :List of 2
  ..$ num: int 2
  ..$ HxW: int [1:2] 459 612
 $ :List of 2
  ..$ num: int 1
  ..$ HxW: int [1:2] 643 857
 $ :List of 2
  ..$ num: int 0
  ..$ HxW: int [1:2] 900 1200

shrunk_original_img <- original_img %>% tf$image$resize(octaves[[1]]$HxW)

img <- original_img
for (octave in octaves) {
  cat(sprintf("Processing octave %i with shape (%s)\n",
             octave$num, paste(octave$HxW, collapse = ", ")))

  img <- img %>%
    tf$image$resize(octave$HxW) %>%
    gradient_ascent_loop(iterations = iterations,
                        learning_rate = step,
                        max_loss = max_loss)

  upscaled_shrunk_original_img <-
    shrunk_original_img %>% tf$image$resize(octave$HxW)

  same_size_original <-
    original_img %>% tf$image$resize(octave$HxW)

  lost_detail <-

```

Загрузка и предварительная обработка тестового изображения

Вычисление целевой формы изображения для различных октав

Сохранение ссылки на исходное изображение (нам нужно сохранить оригинал)

Итерация по разным октавам

Увеличение масштаба изображения

Запуск градиентного восхождения

Увеличение меньшей версии исходного изображения: оно будет пикселизированным

Вычисление высококачественной версии исходного изображения в этом размере

Разница между ними заключается в деталях, которые были потеряны при масштабировании

```

same_size_original - upscaled_shrunk_original_img

img %<% '+'(lost_detail) ←
shrunk_original_img <-
  original_img %>% tf$image$resize(octave$HxW)
}

img <- deprocess_image(img)

img %>% display_image_tensor()

img %>%
  tf$io$encode_png() %>%
  tf$io$write_file("dream.png", .) ← Сохранение окончательного результата

```

Оригинальная сеть Inception V3 была обучена распознавать образы на изображениях размером  $299 \times 299$ , поэтому, учитывая, что процесс выполняет уменьшение масштаба изображений с разумным коэффициентом, реализация DeepDream производит лучшие результаты для изображений с размерами между  $300 \times 300$  и  $400 \times 400$ . Тем не менее вы можете использовать тот же код для обработки изображений любого размера, с любым соотношением сторон.

На графическом процессоре обработка изображения занимает всего несколько секунд. На рис. 12.7 показан результат обработки тестового изображения.

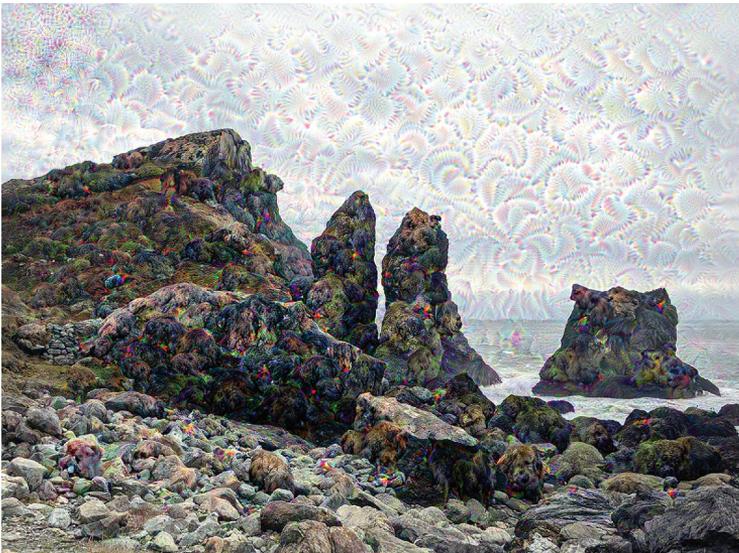


Рис. 12.7 Результаты обработки тестового изображения нашей реализацией DeepDream

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 геометричные 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 показаны результаты, полученные с использованием разных конфигураций слоев из изображения с аппетитной домашней выпечкой.



Рис. 12.8 Примеры различных конфигураций DeepDream на примере изображения домашней выпечки

## 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 зрительные галлюцинации, возникающие у людей, страдающих нарушением работы зрительного отдела коры головного мозга.
- Этот процесс не является специфическим для моделирования изображений или даже для сверточных сетей. Его можно применить к речи, музыке и т. д.

## 12.3 Нейронный перенос стиля

Кроме DeepDream, существует еще одна важная разработка в области изменения изображений с использованием глубокого обучения – *нейронный перенос стиля* (иногда называют *передачей стиля*), представленный Леоном Гатисом с коллегами летом 2015 года<sup>1</sup>. Алгоритм переноса стиля претерпел множество усовершенствований, породил разнообразные вариации и нашел применение в различных приложениях обработки фотографий для смартфонов. Для простоты в этом разделе основное внимание уделяется формулировке из оригинальной статьи.

Перенос стиля заключается в применении стиля изображения-образца к целевому изображению при сохранении содержания целевого изображения. Пример передачи стиля изображен на рис. 12.9.



Рис. 12.9 Пример переноса стиля

В данном контексте под *стилем* в основном подразумеваются текстуры, цветовая палитра и визуальные шаблоны в различных пространственных масштабах, а под *содержанием* – 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 города Тюбингена – это содержание.

Идея переноса стиля, тесно связанная с созданием текстур, давно зрела в сообществе исследователей, увлеченных обработкой изображений, прежде чем воплотилась в алгоритм нейронной передачи стиля в 2015 году. Однако, как оказалось, реализации переноса стиля, основанные на глубоком обучении, не имеют аналогов среди прежних достижений, использовавших классические методики компьютерного зрения, и потому они породили настоящий бум в сфере художественных приложений компьютерного зрения.

В основе реализации переноса стиля лежит та же идея, что занимает центральное положение во всех алгоритмах глубокого обучения: вы задаете функцию потерь, чтобы определить цель для достижения, и минимизируете значение функции. Вы знаете, чего

<sup>1</sup> Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, *A Neural Algorithm of Artistic Style*, arXiv (2015), <https://arxiv.org/abs/1508.06576>.

хотите: сохранить содержание исходного изображения и передать стиль изображения-образца. Определив математически *содержание* 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 содержания

Как вы уже знаете, активации из нижних слоев в сети содержат *локальную* информацию об изображении, тогда как активации из верхних слоев содержат все более *глобальную, абстрактную* информацию. Другими словами, активации разных слоев сверточной сети представляют разложение содержания изображения в разных пространственных масштабах. Поэтому можно ожидать, что содержание более глобального и абстрактного изображения будет захватываться представлениями верхних слоев сети.

Соответственно, хорошим кандидатом на функцию потерь содержания 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 содержание входных изображений, тогда минимизация этой функции может рассматриваться как способ сохранения содержания изображения.

### 12.3.2 Функция потерь стиля

Функция потерь содержания 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 про-

пространственных масштабах, выделяемых сверточной сетью. В качестве функции потерь стиля Гатис с коллегами используют матрицу Грама активаций слоя: внутреннее произведение карт признаков данного слоя. Это внутреннее произведение можно интерпретировать как матрицу корреляций между признаками слоя. Корреляции фиксируют статистики шаблонов определенного пространственного масштаба, которые эмпирически соответствуют текстурам, обнаруженным в этом масштабе.

Следовательно, минимизация функции потерь стиля направлена на сохранение сходных внутренних корреляций между активациями разных слоев изображения-образца и генерируемого изображения. Это, в свою очередь, гарантирует, что текстуры, найденные в разных пространственных масштабах, будут выглядеть одинаково в изображении-образце и в сгенерированном изображении.

Проще говоря, предварительно обученную сверточную сеть можно использовать для определения функции потерь, которая будет делать следующее:

- сохранять содержание, 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 нейронного переноса стиля 2015 года с применением Keras. Как вы увидите далее, он имеет много общего с реализацией DeepDream, представленной в предыдущем разделе.

### 12.3.3 Реализация переноса стиля в Keras

Нейронный перенос стиля можно реализовать с использованием любой обученной сверточной сети. Здесь мы используем сеть VGG19, которую применяли Гатис с коллегами. VGG19 – это упрощенный вариант сети VGG16, представленной в главе 5, с тремя сверточными слоями. Вот как выглядит процесс в целом:

- настройка сети, которая вычисляет активации слоя VGG19 одновременно для изображения-образца, целевого и сгенерированного изображений;
- использование активаций, вычисленных по всем трем изображениям, для определения общей функции потерь, описанной выше, которая будет минимизирована для достижения эффекта передачи стиля;

- настройка процедуры градиентного спуска для минимизации этой функции потерь.

Начнем с определения путей к источникам стиля и содержания. Чтобы гарантировать совместимость размеров обрабатываемых изображений (сильно отличающиеся размеры затрудняют передачу стиля), приведем их к общей высоте в 400 пикселей.

#### Листинг 12.16 Получение изображений-источников стиля и содержания

```
base_image_path <- get_file( ← | Путь к изображению, которое
                             | мы хотим преобразовать
                             "sf.jpg", origin = "https://img-datasets.s3.amazonaws.com/sf.jpg")

style_reference_image_path <- get_file( ← | Путь к изображению, которое
                                           | служит источником стиля
                                           "starry_night.jpg",
                                           origin = "https://img-datasets.s3.amazonaws.com/starry_night.jpg")

c(original_height, original_width) %<-% {
  base_image_path %>%
    tf$io$read_file() %>%
    tf$io$decode_image() %>%
    dim() %>% .[1:2]
}
img_height <- 400
img_width <- round(img_height * (original_width / | Размеры
                                                    | сгенерированного
                                                    | изображения
                                                    original_height))
```

Изображение, которое служит источником содержания, показано на рис. 12.10, а на рис. 12.11 представлен источник стиля.



Рис. 12.10 Изображение – источник содержания: вид на Сан-Франциско из Ноб-Хилла



Рис. 12.11 Изображение – источник стиля: «Звездная ночь» Ван Гога

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

```

      Вспомогательная функция для открытия, изменения размера
      и форматирования изображений в соответствующие массивы
preprocess_image <- function(image_path) {
  image_path %>%
    tf$io$read_file() %>%
    tf$io$decode_image() %>%
    tf$image$resize(as.integer(c(img_height, img_width))) %>%
    k_expand_dims(axis = 1) %>%
    imagenet_preprocess_input()
}
      Добавление размерности по оси пакетов

deprocess_image <- tf_function(function(img) {
  if (length(dim(img)) == 4)
    img <- k_squeeze(img, axis = 1)
  c(b, g, r) %<-% {
    img %>%
      k_reshape(c(img_height, img_width, 3)) %>%
      k_unstack(axis = 3)
  }
}
      Вспомогательная функция
      для преобразования
      тензора в изображение
      Распаковывает по третьей оси
      и возвращает список длиной 3

```

Также принимает изображение с размером по оси пакетов 1.  
(Выдаст ошибку, если первая ось не имеет размера 1)

```

r %<>% `+`(123.68)
g %<>% `+`(103.939)
b %<>% `+`(116.779)
k_stack(c(r, g, b), axis = 3) %>%
  k_clip(0, 255) %>%
  k_cast("uint8")
})

```

Центрирование к нулю путем удаления среднего значения пикселя из ImageNet. Эта операция отменяет преобразование, сделанное `imagenet_preprocess_input()`

Обратите внимание, что мы меняем порядок каналов BGR на RGB. Это также часть обращения `imagenet_preprocess_input()`

### Внутренние функции Keras (`k_*`)

В этой версии `preprocess_image()` и `deprocess_image()` мы используем внутренние функции Keras, такие как `k_expand_dims()`, но в более ранних версиях мы использовали функции из модуля `tf`, такие как `tf$expand_dims()`. Чем они различаются?

Пакет Keras содержит обширный набор внутренних функций, все они начинаются с префикса `k_`. Они являются наследием того времени, когда библиотека Keras предназначалась для работы с несколькими внутренними модулями. Сегодня больше распространен прямой вызов функций в модуле `tf`, где они обычно предоставляют больше полезных возможностей. Одна из прелестей внутренних функций `keras::k_` заключается в том, что все они ведут отсчет от 1 и часто при необходимости выполняют автоматическое приведение аргументов функции к целому числу. Например, команда `k_expand_dims(axis = 1)` эквивалентна `tf$expand_dims(axis = 0L)`.

Разработка внутренних функций Keras в основном прекращена, но они входят в перечень стабильных компонентов TensorFlow, имеют полноценную поддержку и не исчезнут в ближайшее время. Вы можете безопасно использовать такие функции, как `k_expand_dims()`, `k_squeeze()` и `k_stack()`, для выполнения общих операций с тензорами, особенно когда проще использовать соглашение об отсчете с 1. Однако если вы обнаружите, что возможностей внутренних функций недостаточно, смело переходите на прямое использование функций модуля `tf`. Вы можете найти дополнительную документацию о внутренних функциях по адресу <https://keras.rstudio.com/articles/backend.html>.

Настроим сеть VGG19. Как и в примере с DeepDream, мы будем использовать предварительно обученную сверточную сеть для создания модели выделения признаков, которая возвращает активации промежуточных слоев – на этот раз всех слоев в модели.

### Листинг 12.18 Использование предварительно обученной модели VGG19 для выделения признаков

```

model <- application_vgg19(weights = "imagenet",
                           include_top = FALSE)
outputs <- list()
for (layer in model$layers)
  outputs[[layer$name]] <- layer$output

```

Строим модель VGG19 с загруженными готовыми весами ImageNet

Модель, которая возвращает значения активации для каждого целевого слоя  
(как именованный список)

```
feature_extractor <- keras_model(inputs = model$inputs,
                                outputs = outputs)
```

Теперь определим функцию потерь содержания, которая позволит гарантировать сходство представлений целевого и сгенерированного изображений в верхнем уровне сети VGG19.

#### Листинг 12.19 Функция потерь содержания

```
content_loss <- function(base_img, combination_img)
  sum((combination_img - base_img) ^ 2)
```

Далее идет функция потерь стиля. Она использует вспомогательную функцию для вычисления матрицы Грама из входной матрицы: карты корреляций, найденных в матрице оригинальных признаков.

#### Листинг 12.20 Функция потерь стиля

```
gram_matrix <- function(x) {
  n_features <- tf$shape(x)[3]
  x %>%
    tf$reshape(c(-1L, n_features)) %>%
    tf$matmul(t(.), .)
}
# Выход имеет форму (n_features, n_features)
```

← x имеет форму (height, width, features)

← Убираем первые две пространственные оси и сохраняем ось признаков

```
style_loss <- function(style_img, combination_img) {
  S <- gram_matrix(style_img)
  C <- gram_matrix(combination_img)
  channels <- 3
  size <- img_height * img_width
  sum((S - C) ^ 2) /
    (4 * (channels ^ 2) * (size ^ 2))
}
```

К этим двум компонентам потерь добавляется третий: функция *общей потери вариации* (total variation loss), которая оперирует пикселями генерируемого изображения. Она стимулирует пространственную целостность генерируемого изображения, что позволяет избежать появления мозаичного эффекта. Ее можно интерпретировать как регуляризацию потерь.

#### Листинг 12.21 Функция общей потери вариации

```
total_variation_loss <- function(x) {
  a <- k_square(x[, NA:(img_height-1), NA:(img_width-1), ] -
               x[, 2:NA, NA:(img_width-1), ])
  b <- k_square(x[, NA:(img_height-1), NA:(img_width-1), ] -
               x[, NA:(img_height-1), 2:NA, ])
  sum((a + b) ^ 1.25)
}
```

Функция потерь, которую мы должны минимизировать, возвращает среднее взвешенное этих трех компонентов. Для вычисления потери содержимого используется только один верхний уровень `block5_conv2`, а для вычисления потери содержимого используется список уровней, включающий нижние и верхние уровни. Общая потеря вариации прибавляется последней.

В зависимости от используемых изображений с целевым содержимым и образцом стиля может появиться желание настроить коэффициент `content_weight` (определяет вклад потерь содержимого в общую величину потерь). Большее значение `content_weight` обеспечит большее сходство сгенерированного изображения с целевым.

### Листинг 12.22 Определение окончательной функции потерь, подлежащей минимизации

```

style_layer_names <- c(
  "block1_conv1",
  "block2_conv1",
  "block3_conv1",
  "block4_conv1",
  "block5_conv1"
)
content_layer_name <- "block5_conv2"
total_variation_weight <- 1e-6
content_weight <- 2.5e-8
style_weight <- 1e-6

compute_loss <-
  function(combination_image, base_image, style_reference_image) {

    input_tensor <-
      list(base_image,
           style_reference_image,
           combination_image) %>%
      k_concatenate(axis = 1)

    features <- feature_extractor(input_tensor)
    layer_features <- features[[content_layer_name]]
    base_image_features <- layer_features[1, , ]
    combination_features <- layer_features[3, , ]

    loss <- 0
    loss %<>% `+`(
      content_loss(base_image_features, combination_features) *
      content_weight
    )

    for (layer_name in style_layer_names) {
      layer_features <- features[[layer_name]]
      style_reference_features <- layer_features[2, , ]
      combination_features <- layer_features[3, , ]

```

Список слоев для вычисления потерь стиля

Слой для вычисления потерь содержания

Весовой коэффициент потерь содержания

Весовой коэффициент потерь стиля

Начальное значение потерь = 0

Инициализируем потерю равной 0

Прибавляем потерю содержания

```

    loss %<>% `+`(
      style_loss(style_reference_features, combination_features) *
      style_weight / length(style_layer_names)
    )
  }

  loss %<>% `+`(
    total_variation_loss(combination_image) *
    total_variation_weight
  )
  loss
}

```

Прибавляем потерю стиля для каждого слоя стиля

Прибавляем общую потерю вариации

Возвращаем сумму потерь содержания, стиля и вариации

Наконец, настроим процесс градиентного спуска. В оригинальной статье Гатиса оптимизация выполняется с использованием алгоритма L-BFGS, но он недоступен в TensorFlow, поэтому здесь мы выполним мини-пакетный градиентный спуск с оптимизатором SGD. Мы будем использовать новую функцию оптимизатора, с которой вы еще незнакомы: *переменный коэффициент обучения* (learning-rate schedule). Мы будем использовать ее для постепенного снижения скорости обучения с очень высокого значения (100) до гораздо меньшего конечного значения (около 20). Благодаря переменной скорости мы добьемся быстрого прогресса на ранних этапах обучения, а затем будем двигаться более осторожно по мере приближения к минимуму потерь.

### Листинг 12.23 Настройка процесса градиентного спуска

```

compute_loss_and_grads <- tf_function(
  function(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)
    list(loss, grads)
  })

optimizer <- optimizer_sgd(
  learning_rate_schedule_exponential_decay(
    initial_learning_rate = 100, 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))

output_dir <- fs::path("style-transfer-generated-images")

```

Мы ускоряем шаг обучения, скомпилировав его как tf\_function()

Мы начинаем с коэффициента обучения 100 и уменьшаем его на 4 % каждые 100 шагов

Используем tf\$Variable() для хранения комбинированного изображения, потому что будем обновлять его во время обучения

```

iterations <- 4000
for (i in seq(iterations)) {
  c(loss, grads) %<-% compute_loss_and_grads(
    combination_image, base_image, style_reference_image)

  optimizer$apply_gradients(list( ← Обновляем комбинированное
    tuple(grads, combination_image))) ← изображение в направлении, которое
                                        уменьшает потери при переносе стиля
  if ((i % 100) == 0) {
    cat(sprintf("Iteration %i: loss = %.2f\n", i, loss))
    img <- deprocess_image(combination_image)
    display_image_tensor(img)
    fname <- sprintf("combination_image_at_iteration_%04i.png", i)
    tf$io$write_file(filename = output_dir / fname, ←
      contents = tf$io$encode_png(img))
  }
}

```

Сохраняем комбинированное изображение  
через равные промежутки времени

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 содержанием не требуют различения мелких деталей, чтобы их можно было опознать. Этот прием не наделен возможностями абстрагирования – с его помощью едва ли получится перенести стиль с одного портрета в другой. Данный алгоритм ближе к классической обработке сигналов, чем к ИИ, поэтому не нужно ожидать от него чего-то сверхъестественного!



Рис. 12.12 Результат переноса стиля

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. Быстрого переноса стиля можно достичь, если сначала потратить много времени на создание входных/выходных обучающих примеров для фиксированного изображения с образцом стиля, используя метод, описанный здесь, а затем обучить простую сверточную сеть данному конкретному преобразованию стиля. После этого можно будет почти мгновенно стилизовать любое изображение: для этого потребуется просто пропустить его через эту маленькую сверточную сеть.

### 12.3.4 Подведение итогов

- Перенос стиля заключается в создании нового изображения, сохраняющего содержимое целевого изображения и оформленного в стиле изображения-образца.
- Содержание может сохраняться активациями верхнего слоя сверточной сети.
- Стиль может сохраняться внутренними корреляциями активаций разных слоев.
- Таким образом, перенос 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 Генерация изображений с помощью вариационных автокодировщиков

Выбор шаблонов из скрытого пространства изображений для создания совершенно новых или редактирования существующих изображений в настоящее время является самым успешным и популярным применением художественных возможностей ИИ.

В этом и в следующем разделах мы рассмотрим некоторые высокоуровневые понятия, связанные с генерацией изображений, а также детали реализации двух основных методов, используемых в этой области: *вариационных автокодировщиков* (Variational AutoEncoders, VAE) и *генеративно-сопоставительных 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

Ключевая 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). Новые изображения занимают промежуточное положение между обучающими изображениями.

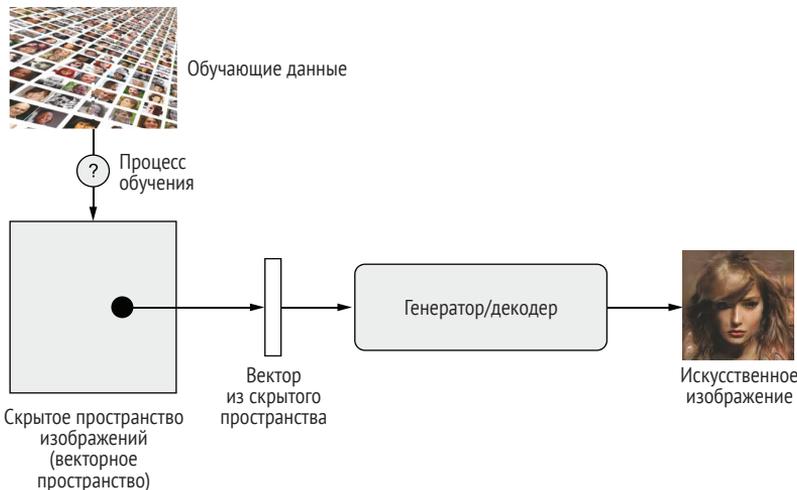


Рис. 12.13 Обученное скрытое векторное пространство изображений и его использование для создания новых изображений

Генеративно-сопоставительные 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 (рис. 12.14). Генеративно-сопоставительные сети

генерируют изображения, потенциально более реалистичные, но скрытое пространство, из которого они исходят, может не обладать структурированностью и непрерывностью.

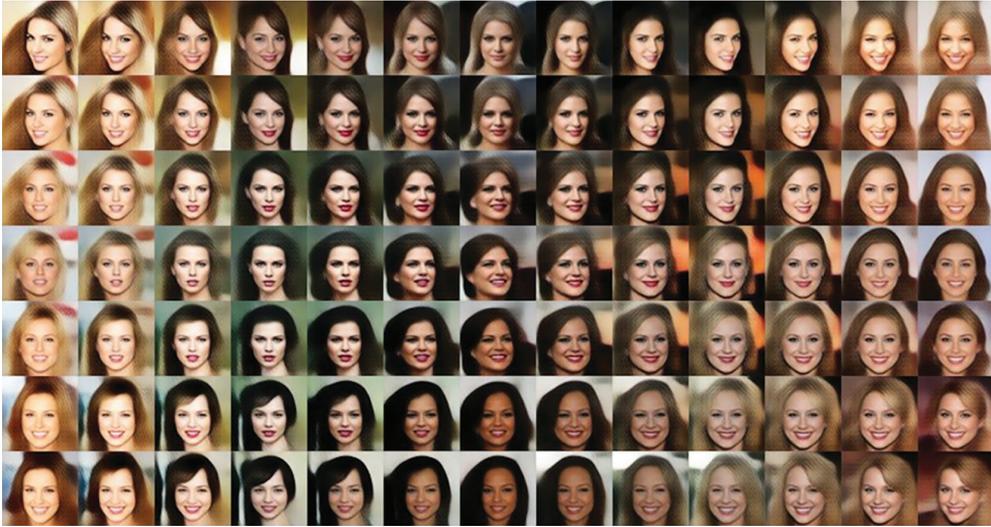


Рис. 12.14 Непрерывное пространство лиц, сгенерированное Томом Уайтом с использованием VAE

### 12.4.2 Концептуальные векторы для редактирования изображений

Мы уже обсуждали идею *концептуальных* (или *понятийных*) *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 (Tom White) из школы дизайна в Университете Виктории (Victoria University School of Design)

в Новой Зеландии, с использованием VAE, обученных на наборах изображений лиц знаменитостей (набор данных CelebA).



Рис. 12.15 Концептуальный вектор «улыбка»

### 12.4.3 Вариационные автокодировщики

Вариационные автокодировщики, открытые одновременно Дидериком Кингма 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 генеративной модели, которая особенно хорошо подходит для редактирования изображений посредством концептуальных векторов. Они представляют современный подход к автокодировщикам – разновидности сетей, целью которых является кодирование входного малоразмерного скрытого пространства 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 модуль-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

<sup>1</sup> Diederik P. Kingma and Max Welling, *Auto-Encoding Variational Bayes*, arXiv (2013), <https://arxiv.org/abs/1312.6114>.

<sup>2</sup> Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra, *Stochastic Backpropagation and Approximate Inference in Deep Generative Models*, arXiv (2014), <https://arxiv.org/abs/1401.4082>.

кодировщик действует как инструмент сжатия входных данных, генерируя на выходе меньший объем информации.

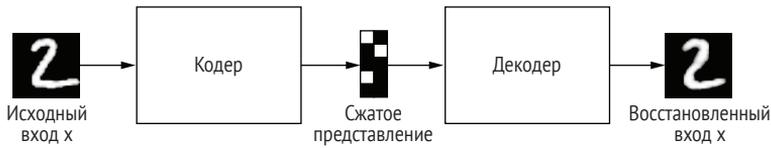


Рис. 12.16 Автокодировщик отображает вход  $x$  в сжатое представление, которое затем декодируется обратно как  $x'$

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 автокодировщики толику статистического волшебства, что заставляет их извлекать непрерывные высокоструктурированные скрытые пространства. Они оказались мощным инструментом для генерирования изображений.

Вместо сжатия в фиксированный код в скрытом пространстве вариационный кодировщик превращает входное изображение в параметры статистического распределения: среднее и дисперсию. По сути, это означает предположение, что входное изображение было сгенерировано статистическим процессом и что случайную составляющую этого процесса необходимо учитывать в ходе кодирования и декодирования. Вариационный автокодировщик затем использует среднее и дисперсию как параметры для случайного отбора одного элемента из распределения и декодирует его обратно в 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.

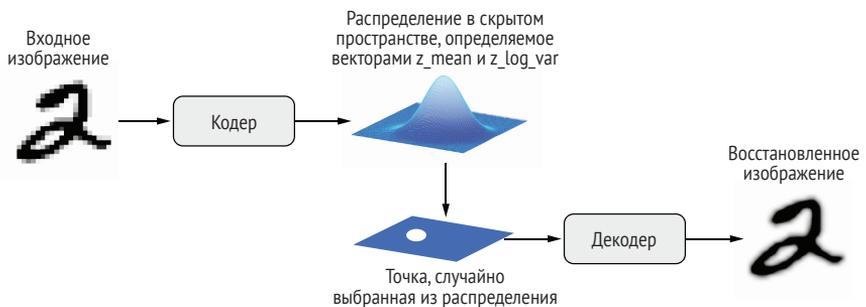


Рис. 12.17 Вариационный автокодировщик отображает изображение в два вектора,  $z\_mean$  и  $z\_log\_sigma$ , которые определяют распределение вероятности в скрытом пространстве, используемое для выбора точки для декодирования

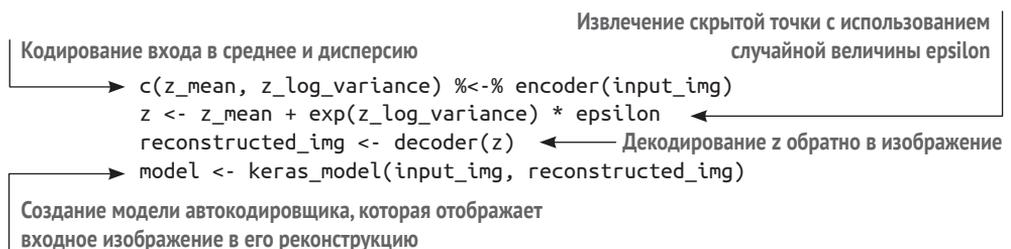
Вот как работает вариационный автокодировщик с технической точки зрения:

- 1 модуль кодировщика превращает выборки из входного 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.

Поскольку `epsilon` является случайным тензором, процесс гарантирует, что каждая точка, близкая к скрытому местоположению, где закодировано `input_img(z_mean)`, может быть декодирована в нечто, похожее на `input_img`, что обеспечивает непрерывную значимость скрытого пространства.

Любые две близкие точки в скрытом пространстве будут декодированы в очень похожие изображения. Непрерывность в сочетании с малой размерностью скрытого пространства заставляет каждое направление в скрытом пространстве кодировать значимую ось изменений данных, что делает скрытое пространство высокоструктурированным и прекрасно подходящим для манипуляций посредством концептуальных векторов.

Параметры вариационного автокодировщика обучаются на двух функциях потерь: *потерях восстановления* (*reconstruction loss*), 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. В общих чертах процесс выглядит так:



Затем можно обучить модель, используя потери восстановления и потери регуляризации. Для вычисления потерь при регуляризации мы обычно используем расхождение Кульбака–Лейблера, предназначенное для приведения распределения выходных данных кодировщика к плавно скругленному нормальному распределению с центром вокруг 0. Это дает кодировщику разумное предположение о структуре скрытого пространства, которое он моделирует.

Далее мы рассмотрим, как выглядит реализация VAE на практике.

### 12.4.4 Реализация VAE с помощью Keras

Мы создадим вариационный автокодировщик, который может генерировать цифры MNIST. Он будет состоять из трех частей:

- сеть кодировщика, которая превращает реальное изображение в среднее значение и дисперсию в скрытом пространстве;
- слой выборки, который берет среднее значение и дисперсию и использует их для выборки случайной точки из скрытого пространства;
- сеть декодера, которая превращает точки из скрытого пространства обратно в изображения.

В листинге 12.24 представлена 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  $z\_log\_var$ . Важная деталь заключается в том, что для уменьшения выборки карт объектов вместо объединения по максимальному из соседних карт используем шаги. В последний раз мы делали это в примере с сегментацией изображения в главе 9. Мы установили, что в целом шаги предпочтительнее, чем объединение по максимальному соседнему для любой модели, которая использует информацию о пространственном расположении признака на изображении. Сейчас это важно, потому что кодировщик должен будет создать закодированное представление изображения, которое можно использовать для восстановления осмысленного изображения.

**Листинг 12.24** Сеть кодировщика VAE

```
latent_dim <- 2 ←————— Размерность скрытого пространства: двумерная плоскость

encoder_inputs <- layer_input(shape = c(28, 28, 1))

x <- encoder_inputs %>%
  layer_conv_2d(32, 3, activation = "relu", strides = 2,
  ↳ padding = "same") %>%
  layer_conv_2d(64, 3, activation = "relu", strides = 2,
  ↳ padding = "same") %>%
  layer_flatten() %>%
  layer_dense(16, activation = "relu")
  Входное изображение
  кодируется в эти два параметра

z_mean <- x %>% layer_dense(latent_dim, name = "z_mean")
z_log_var <- x %>% layer_dense(latent_dim, name = "z_log_var")

encoder <- keras_model(encoder_inputs, list(z_mean, z_log_var),
  name = "encoder")
```

Так выглядит краткая сводка кодировщика:

encoder

Model: "encoder"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 28, 28, 1)]	0	[]
conv2d_1 (Conv2D)	(None, 14, 14, 32)	320	['input_1[0][0]']
conv2d (Conv2D)	(None, 7, 7, 64)	18496	['conv2d_1[0][0]']
flatten (Flatten)	(None, 3136)	0	['conv2d[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]']

Total params: 69,076  
 Trainable params: 69,076  
 Non-trainable params: 0

Далее в листинге 12.25 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**

```
layer_sampler <- new_layer_class(
  classname = "Sampler",
  call = function(z_mean, z_log_var) {
    epsilon <- tf$random$normal(shape = tf$shape(z_mean))
    z_mean + exp(0.5 * z_log_var) * epsilon
  }
)
```

Применяем формулу выборки VAE →

z\_mean и z\_log\_var здесь оба будут иметь форму (batch\_size, latent\_dim), например (128, 2)

Выбираем пакет произвольных нормальных векторов

В листинге 12.26 показана реализация 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 <- layer_input(shape = c(latent_dim))
decoder_outputs <- latent_inputs %>%
  layer_dense(7 * 7 * 64, activation = "relu") %>%
  layer_reshape(c(7, 7, 64)) %>%
```

→ Подаем z на вход

← Обращаем слой layer\_flatten() кодировщика

→ Задаем такое же количество коэффициентов, которое у нас было на уровне слоя Flatten в кодировщике

```

layer_conv_2d_transpose(64, 3, activation = "relu",
                        strides = 2, padding = "same") %>%
layer_conv_2d_transpose(32, 3, activation = "relu",
                        strides = 2, padding = "same") %>%
layer_conv_2d(1, 3, activation = "sigmoid",
              padding = "same") ← Выход имеет форму (28, 28, 1)
decoder <- keras_model(latent_inputs, decoder_outputs,
                       name = "decoder")

```

Обращаем слой `layer_conv_2d()`  
кодировщика

Так выглядит краткая сводка декодера:

decoder

Model: "decoder"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 2)]	0
dense_1 (Dense)	(None, 3136)	9408
reshape (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 14, 14, 64)	36928
conv2d_transpose (Conv2DTranspose)	(None, 28, 28, 32)	18464
conv2d_2 (Conv2D)	(None, 28, 28, 1)	289

Total params: 65,089

Trainable params: 65,089

Non-trainable params: 0

Теперь можно собрать из компонентов полную модель вариационного автокодировщика. Это первый пример модели, которая не выполняет обучение с учителем (автокодировщик относится к *самообучаемым моделям*, поскольку в качестве целей он использует свои входные данные). В случае отказа от традиционного обучения с учителем обычно приходится создавать новый класс модели `new_model_class()` и разрабатывать собственный цикл обучения `train_step()`, в которых определена новая логика обучения – рабочий процесс, о котором вы узнали в главе 7. Этим мы и займемся дальше.

### Листинг 12.27 Модель VAE с собственным циклом обучения `train_step()`

```

model_vae <- new_model_class(
  classname = "VAE",

  initialize = function(encoder, decoder, ...) {
    super$initialize(...)
    self$encoder <- encoder ← Указываем self вместо private, потому что
    self$decoder <- decoder   хотим, чтобы веса слоев автоматически
    self$sampler <- layer_sampler() ← отслеживались базовым классом модели Keras
    self$total_loss_tracker <-

```

```

metric_mean(name = "total_loss")
self$reconstruction_loss_tracker <-
  metric_mean(name = "reconstruction_loss")
self$kl_loss_tracker <-
  metric_mean(name = "kl_loss")
},

```

Используем эти метрики для отслеживания средних потерь за каждую эпоху

```

metrics = mark_active(function() {
  list(
    self$total_loss_tracker,
    self$reconstruction_loss_tracker,
    self$kl_loss_tracker
  )
}),

```

Перечисляем метрики в текущих свойствах, чтобы платформа могла сбрасывать их после каждой эпохи (или между несколькими вызовами `fit()/evaluate()`)

```

train_step = function(data) {
  with(tf$GradientTape() %as% tape, {
    c(z_mean, z_log_var) %<% self$encoder(data)
    z <- self$sampler(z_mean, z_log_var)

    reconstruction <- decoder(z)
    reconstruction_loss <-
      loss_binary_crossentropy(data, reconstruction) %>%
      sum(axis = c(2, 3)) %>%
      mean()

```

Суммируем потери восстановления по пространственным измерениям (вторая и третья оси) и берем их среднее значение по размеру пакета

Берем среднее значение общих потерь в пакете

Общие потери для каждого случая в пакете; сохраняем ось пакетов

```

    kl_loss <- -0.5 * (1 + z_log_var - z_mean^2 - exp(z_log_var))
    total_loss <- reconstruction_loss + mean(kl_loss)
  })

```

Прибавляем член регуляризации (расстояние Кульбака-Лейблера)

```

  grads <- tape$gradient(total_loss, self$trainable_weights)
  self$optimizer$apply_gradients(zip_lists(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)

  list(total_loss = self$total_loss_tracker$result(),
        reconstruction_loss = self$reconstruction_loss_tracker$result(),
        kl_loss = self$kl_loss_tracker$result())
}
)

```

Наконец, мы готовы создать и обучить модель на наборе MNIST (листинг 12.28). Поскольку вычислением потерь у нас занимается собственный слой, мы не указываем внешнюю функцию потерь на этапе компиляции (`loss = NULL`). 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 вариационного автокодировщика

```

library(listarrays)
c(c(x_train, .), c(x_test, .)) %<-% dataset_mnist()

mnist_digits <-
  bind_on_rows(x_train, x_test) %>%
  expand_dims(-1) %>%
  { . / 255 }

str(mnist_digits)

num [1:70000, 1:28, 1:28, 1] 0 0 0 0 0 0 0 0 0 0 ...

vae <- model_vae(encoder, decoder)
vae %>% compile(optimizer = optimizer_adam())
vae %>% fit(mnist_digits, epochs = 30, batch_size = 128)

```

Предоставляем `bind_on_rows()` и другие функции для управления массивами R

Мы обучаем модель на всех цифрах MNIST, поэтому объединяем обучающую и проверочную выборки по оси пакетов

Мы не передаем аргумент потери в `compile()`, потому что потеря уже является частью `train_step()`

Мы не передаем цели в `fit()`, потому что `train_step()` их не ожидает

После обучения модели мы можем использовать сеть `decoder` для преобразования произвольных векторов скрытого пространства в изображения.

## Листинг 12.29 Выбор сетки с точками из двумерного скрытого пространства и их декодирование в изображения

```

n <- 30
digit_size <- 28
z_grid <-
  seq(-1, 1, length.out = n) %>%
  expand_grid(., .) %>%
  as.matrix()
decoded <- predict(vae$decoder, z_grid)

z_grid_i <- seq(n) %>% expand_grid(x = ., y = .)
figure <- array(0, c(digit_size * n, digit_size * n))
for (i in 1:nrow(z_grid_i)) {
  c(xi, yi) %<-% z_grid_i[i, ]
  digit <- decoded[i, , ]
  figure[seq(to = (n + 1 - xi) * digit_size, length.out = digit_size),
    seq(to = yi * digit_size, length.out = digit_size)] <-
    digit
}

par(pty = "s")
lim <- extendrange(r = c(-1, 1),
  f = 1 - (n / (n+.5)))

```

Создаем двумерную сетку линейно расположенных образцов

Преобразовываем декодированные цифры с формой (900, 28, 28, 1) в массив R с формой (28\*30, 28\*30) для вывода на экран

Получаем декодированные цифры

Будем отображать сетку 30x30 цифр (900 изображений)

Квадратный шаблон

Расширяем `lim` так, чтобы точка (-1, 1) находилась в центре цифры

```

plot(NULL, frame.plot = FALSE,
      ylim = lim, xlim = lim,
      xlab = ~z[1], ylab = ~z[2])
rasterImage(as.raster(1 - figure, max = 1),
            lim[1], lim[1], lim[2], lim[2],
            interpolate = FALSE)

```

Вычитаем 1 для инверсии цветов

Передаем объект формулы в xlab для формирования правильной подписи

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 наделяются определенным смыслом: например, есть направление «пятерочности», «единичности» и т. д.

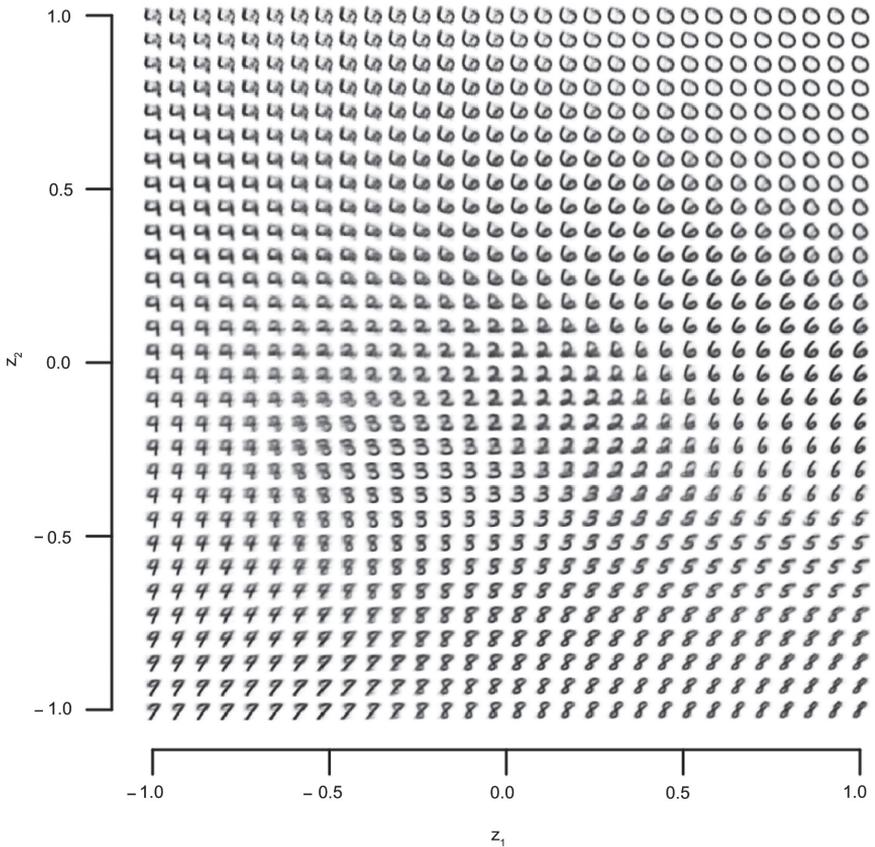


Рис. 12.18 Сетка цифр, декодированных из скрытого пространства

В следующем разделе мы подробно рассмотрим один важный инструмент создания искусственных изображений: генеративно-сопоставительные сети.

### 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 увидеть прежде не встречавшиеся изображения. Существует два основных инструмента для решения этой задачи: вариационные автокодировщики (VAE) и генеративно-сопоставительные 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 мультимпликации путем прохождения через раздел скрытого пространства, когда начальное изображение постепенно и непрерывно преобразуется в другие изображения.
- Генеративно-сопоставительные 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-сопоставительные сети пользуются очень большой популярностью в академической среде. В следующем разделе вы узнаете, как они работают и как их реализовать.

## 12.5 Введение в генеративно-сопоставительные сети

*Генеративно-сопоставительные сети* (Generative Adversarial Networks, GAN), впервые представленные в 2014 году Яном Гудфеллоу и его коллегами<sup>1</sup>, – альтернатива вариационным автокодировщикам, позволяющая выделять скрытые пространства изображений. Они позволяют генерировать очень реалистичные искусственные изображения, статистически не отличимые от настоящих.

Чтобы проще было понять суть генеративно-сопоставительной 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 мастер-

---

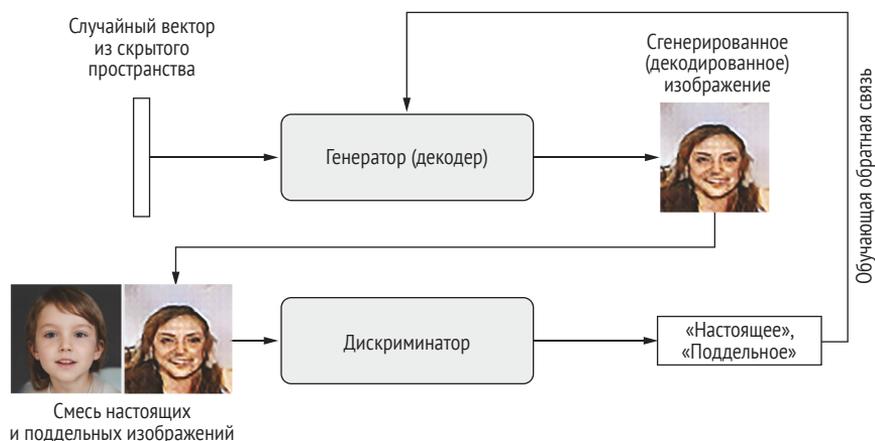
<sup>1</sup> Ian Goodfellow et al., *Generative Adversarial Networks*, arXiv (2014), <https://arxiv.org/abs/1406.2661>.

скую 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.

В этом и заключается суть генеративно-состязательной сети. 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 это изображение из обучающего набора или сгенерировано сетью-генератором.

Сеть-генератор обучается обманывать сеть-дискриминатор и, соответственно, учится создавать все более реалистичные изображения: поддельные изображения, не отличимые от настоящих в той мере, на какую способна сеть-дискриминатор (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.



**Рис. 12.19** Генератор преобразует случайные скрытые векторы в изображения, а дискриминатор стремится отличить настоящие изображения от сгенерированных искусственно. Генератор обучается обманывать дискриминатор

Примечательно, что генеративно-сопоставительная 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-сопоставительные сети трудно поддаются обучению – чтобы получить действующую генеративно-сопоставительную сеть, требуется приложить большие усилия по настройке архитектуры модели и параметров обучения.

### 12.5.1 Реализация генеративно-сопоставительной сети

В этом разделе мы расскажем, как реализовать простейшую генеративно-сопоставительную сеть с использованием Keras, потому что сети этого вида очень сложны и глубокое погружение в технические детали архитектур, подобных архитектуре StyleGAN2, сгенерировавшей изображения на рис. 12.20, выходит далеко за рамки этой книги. В следующем примере мы будем использовать *глубокую сверточную GAN* (Deep Convolutional GAN, DCGAN), в которой генератор и дискриминатор являются глубокими сверточными сетями.

Мы будем обучать нашу GAN на большом наборе изображений CelebFaces Attributes (известном также как CelebA). Это набор из 200 000 изображений лиц знаменитостей (<http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>). Чтобы ускорить обучение, мы уменьшим размер изображений до 64×64, поэтому обученная модель будет генерировать изображения человеческих лиц размером 64×64. В общих чертах GAN выглядит примерно так:

- сеть `generator` отображает векторы с формой `(latent_dim)` 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  $gan(x) = discriminator(generator(x))$ . То есть сеть `gan` 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 мы используем градиенты весов генератора в отношении потерь модели `gan`. То есть на каждом шаге мы будем смещать веса генератора в направлении увеличения вероятности классификации дискриминатором изображений,

декодированных генератором как «настоящие». Иными словами, мы будем обучать генератор обманывать дискриминатор.



Рис. 12.20 «Обитатели» скрытого пространства. Изображения, созданные проектом <https://thispersondoesnotexist.com> с использованием модели StyleGAN2. (Изображение предоставлено автором веб-сайта Филиппом Ваном. Использована модель StyleGAN2, авторы Каррас и др., <https://arxiv.org/abs/1912.04958>)

## 12.5.2 Полезные технические приемы

Процесс обучения и настройки генеративно-сопоставительных 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 полезных технических приемов и хитростей, применяемых при реализации генератора и дискриминатора GAN в этом разделе. Это не полный список; еще множество хитростей, имеющих отношение к GAN, можно найти в специализированной литературе:

- мы используем шаги вместо объединения для понижения разрешения карт признаков в дискриминаторе, как мы это делали в нашем кодировщике 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-сопоставительные сети легко могут застревать на разных препятствиях. Введение случайной составляющей в процесс обучения помогает предотвратить это. Мы вводим случайный компонент двумя способами: используя прореживание в дискриминаторе и добавляя случайный шум в метки для дискриминатора;

- разреженные градиенты могут препятствовать обучению GAN. В глубоком обучении разреженность часто является желательным свойством, но не в случае с GAN. Разреженность градиента могут вызывать операции выбора максимального значения по соседним элементам (max pooling) и активации `relu`. Вместо выбора максимального значения для уменьшения разрешения мы рекомендуем использовать свертки с шагом, а вместо функции активации `relu` – слой `layer_activation_leaky_relu()`. 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 `layer_conv_2d_transpose()` или `layer_conv_2d()` в генераторе и дискриминаторе.

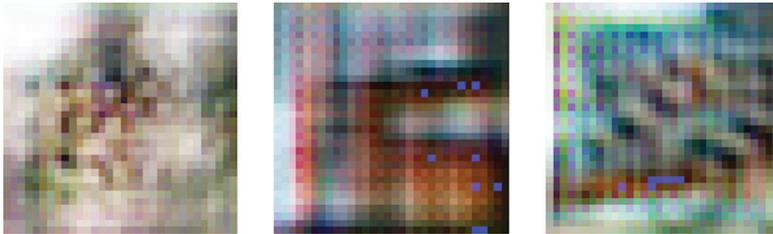


Рис. 12.21 Артефакты типа «шахматная доска», вызванные несовпадением размеров шага и ядра, из-за чего происходит неравномерный охват пространства пикселей: одна из многих особенностей GAN, доставляющих хлопоты

### 12.5.3 Получение набора данных CelebA

Вы можете загрузить набор данных вручную с веб-сайта <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>. Поскольку набор данных размещен на Google Drive, вы также можете загрузить его с помощью утилиты `gdown`:

```
reticulate::py_install("gdown", pip = TRUE) ← Установка gdown
system("gdown 107m1010EJjLE5QxLZiM9Fpjs70j6e684")
```

```
Downloading...
From: https://drive.google.com/uc?id=107m1010EJjLE5QxLZiM9Fpjs70j6e684
To: img_align_celeba.zip
32%|██████████          | 467M/1.44G [00:13<00:23, 41.3MB/s]
```

Скачивание скачтых данных при помощи `gdown`

После загрузки архива данных распакуйте его в папку `celeba_gan`.

**Листинг 12.30** Получение данных CelebA

```
zip::unzip("img_align_celeba.zip", exdir = "celeba_gan")
```

→ Распаковка данных

Получив папку с несжатými изображениями, дальше можно применить `image_dataset_from_directory()`, чтобы превратить ее в набор данных TensorFlow. Поскольку нам нужны только изображения (меток нет), зададим параметр преобразования `label_mode = NULL`.

**Листинг 12.31** Создание набора данных TensorFlow из папки с изображениями

```
dataset <- image_dataset_from_directory(
  "celeba_gan",
  label_mode = NULL,
  image_size = c(64, 64),
  batch_size = 32,
  crop_to_aspect_ratio = TRUE
)
```

Возвращаем только изображения без меток

Изменяем размер изображений до 64×64, используя разумную комбинацию обрезки и изменения размера, чтобы сохранить соотношение сторон. Нам нужно сохранить пропорции лиц!

Наконец, масштабируем изображения к диапазону [0-1] (листинг 12.32).

**Листинг 12.32** Масштабирование изображений

```
library(tfdatasets)
dataset %<>% dataset_map(~ .x / 255)
```

Для просмотра примера изображения можно воспользоваться кодом из листинга 12.33.

**Листинг 12.33** Просмотр первого изображения

```
x <- dataset %>% as_iterator() %>% iter_next()
display_image_tensor(x[1, , , ], max = 1)
```



## 12.5.4 Дискриминатор

Начнем с разработки модели `discriminator`, которая получает в качестве входных данных изображение-кандидат (реальное или искусственное) и относит его к одному из двух классов: «подделка» или «настоящее, имеющееся в обучающем наборе». Одна из многих проблем, которые обычно возникают с GAN, заключается в том, что генератор может выдавать изображения, которые выглядят как шум. Возможное решение – использовать прореживание в дискриминаторе, что мы и сделаем (листинг 12.34).

**Листинг 12.34** Сеть дискриминатора GAN

```
discriminator <-
  keras_model_sequential(name = "discriminator",
    input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(64, kernel_size = 4, strides = 2, padding = "same") %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_conv_2d(128, kernel_size = 4, strides = 2, padding = "same") %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_conv_2d(128, kernel_size = 4, strides = 2, padding = "same") %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_flatten() %>%
  layer_dropout(0.2) %>%
  layer_dense(1, activation = "sigmoid")
```

Один слой прореживания: очень важная хитрость!

Так выглядит краткая сводка модели дискриминатора:

```
discriminator
```

```
Model: "discriminator"
```

Layer (type)	Output Shape	Param #
conv2d_99 (Conv2D)	(None, 32, 32, 64)	3136
leaky_re_lu_2 (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_98 (Conv2D)	(None, 16, 16, 128)	131200
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_97 (Conv2D)	(None, 8, 8, 128)	262272
leaky_re_lu (LeakyReLU)	(None, 8, 8, 128)	0
flatten_1 (Flatten)	(None, 8192)	0
dropout (Dropout)	(None, 8192)	0
dense_7 (Dense)	(None, 1)	8193

=====  
 Total params: 404,801  
 Trainable params: 404,801  
 Non-trainable params: 0

## 12.5.5 Генератор

Далее разработаем модель генератор, которая превращает вектор (из скрытого пространства – во время обучения он будет выбран случайным образом) в изображение-кандидат.

**Листинг 12.35** Сеть генератора GAN

```
latent_dim <- 128
generator <-
  keras_model_sequential(name = "generator",
    input_shape = c(latent_dim)) %>%
  layer_dense(8 * 8 * 128) %>%
  layer_reshape(c(8, 8, 128)) %>%
  layer_conv_2d_transpose(128, kernel_size = 4,
    strides = 2, padding = "same") %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_conv_2d_transpose(256, kernel_size = 4,
    strides = 2, padding = "same") %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_conv_2d_transpose(512, kernel_size = 4,
    strides = 2, padding = "same") %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_conv_2d(3, kernel_size = 5, padding = "same",
    activation = "sigmoid")
```

← Скрытое пространство будет состоять из 128-мерных векторов

← Производим такое же количество коэффициентов, которое у нас было на уровне слоя Flatten в кодировщике

← Обращаем layer\_conv\_2d() кодировщика

← Обращаем layer\_flatten() кодировщика

← Используем Leaky Relu в качестве функции активации

Так выглядит краткая сводка модели генератора:

generator

Model: "generator"

Layer (type)	Output Shape	Param #
dense_8 (Dense)	(None, 8192)	1056768
reshape_1 (Reshape)	(None, 8, 8, 128)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 16, 16, 128)	262272
leaky_re_lu_5 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 32, 32, 256)	524544
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 256)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 64, 64, 512)	2097664
leaky_re_lu_3 (LeakyReLU)	(None, 64, 64, 512)	0
conv2d_100 (Conv2D)	(None, 64, 64, 3)	38403
=====		
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.

Вкратце, одна полная эпоха обучения выглядит следующим образом:

- 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 генератор, используя эти случайные векторы, с целями, которые всегда говорят: «это настоящие изображения». Это приведет к смещению весов генератора в направлении, увеличивающем вероятность получить от дискриминатора ответ «настоящее» для сгенерированных изображений; таким образом генератор научится обманывать дискриминатор.

Реализуем этот процесс в коде. Как и в предыдущем примере с VAE, мы будем использовать новый класс модели `new_model_class()` с разработанным заново циклом обучения `train_step()`. Нам понадобятся два оптимизатора (один для генератора и один для дискриминатора), поэтому мы также переопределим метод `compile()`, чтобы разрешить передачу двух оптимизаторов.

### Листинг 12.36 Модель GAN

```
GAN <- new_model_class(
  classname = "GAN",

  initialize = function(discriminator, generator, latent_dim) {
    super$initialize()
    self$discriminator <- discriminator
    self$generator <- generator
    self$latent_dim <- as.integer(latent_dim)
```

```

        Настройка метрик для отслеживания
        двух потерь в каждую эпоху обучения
self$d_loss_metric <- metric_mean(name = "d_loss")
self$g_loss_metric <- metric_mean(name = "g_loss")
},

compile = function(d_optimizer, g_optimizer, loss_fn) {
  super$compile()
  self$d_optimizer <- d_optimizer
  self$g_optimizer <- g_optimizer
  self$loss_fn <- loss_fn
},

metrics = mark_active(function() {
  list(self$d_loss_metric,
        self$g_loss_metric)
}),

train_step = function(real_images) {
  batch_size <- tf$shape(real_images)[1]
  random_latent_vectors <-
    tf$random$normal(shape = c(batch_size, self$latent_dim))
  generated_images <-
    self$generator(random_latent_vectors)
  combined_images <-
    tf$concat(list(generated_images,
                   real_images),
              axis = 0L)
  labels <-
    tf$concat(list(tf$ones(tuple(batch_size, 1L)),
                  tf$zeros(tuple(batch_size, 1L))),
              axis = 0L)
  labels %<>% `+`(
    tf$random$uniform(tf$shape(.), maxval = 0.05))
  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_lists(grads, self$discriminator$trainable_weights))
  random_latent_vectors <-
    tf$random$normal(shape = c(batch_size, self$latent_dim))
  misleading_labels <- tf$zeros(tuple(batch_size, 1L))
  with(tf$GradientTape() %as% tape, {
    predictions <- random_latent_vectors %>%

```

Вызов train\_step для пакета настоящих изображений

Выборка случайных точек в скрытом пространстве

Декодируем их для создания поддельных изображений

Объединяем поддельные изображения с настоящими

Добавление меток, отличающих настоящие изображения от поддельных

Добавление случайного шума к меткам – это важная хитрость!

Обучаем дискриминатор

Выбор произвольных точек из скрытого пространства

Присоединяем метку «настоящее изображение» (это ложь!)

```

        self$generator() %>%
        self$discriminator()
    g_loss <- self$loss_fn(misleading_labels, predictions)
  })
  grads <- tape$gradient(g_loss, self$generator$trainable_weights)
  self$g_optimizer$apply_gradients(
    zip_lists(grads, self$generator$trainable_weights))

  self$d_loss_metric$update_state(d_loss)
  self$g_loss_metric$update_state(g_loss)

  list(d_loss = self$d_loss_metric$result(),
       g_loss = self$g_loss_metric$result())
})

```

Обучаем  
генератор

Прежде чем начать обучение, настроим обратный вызов для отслеживания результатов: он будет использовать генератор для создания и сохранения ряда искусственных изображений в конце каждой эпохи.

### Листинг 12.37 Обратный вызов, который отбирает сгенерированные изображения во время обучения

```

callback_gan_monitor <- new_callback_class(
  classname = "GANMonitor",

  initialize = function(num_img = 3, latent_dim = 128,
                        dirpath = "gan_generated_images") {
    private$num_img <- as.integer(num_img)
    private$latent_dim <- as.integer(latent_dim)
    private$dirpath <- fs::path(dirpath)
    fs::dir_create(dirpath)
  },

  on_epoch_end = function(epoch, logs = NULL) {
    random_latent_vectors <-
      tf$random$normal(shape = c(private$num_img, private$latent_dim))

    generated_images <- random_latent_vectors %>%
      self$model$generator() %>%
      { tf$saturate_cast(. * 255, "uint8") }

    for (i in seq(private$num_img))
      tf$io$write_file(
        filename = private$dirpath / sprintf("img_%03i_%02i.png",
        epoch, i),
        contents = tf$io$encode_png(generated_images[i, , , ])
      )
  }
)

```

Масштабируем и урезаем до диапазона  
[0, 255] и приводим к uint8

Наконец, мы можем приступить к обучению.

**Листинг 12.38** Компиляция и обучение GAN

```

epochs <- 100
gan <- GAN(discriminator = discriminator,
           generator = generator,
           latent_dim = latent_dim)
gan %>% compile(
  d_optimizer = optimizer_adam(learning_rate = 0.0001),
  g_optimizer = optimizer_adam(learning_rate = 0.0001),
  loss_fn = loss_binary_crossentropy()
)
gan %>% fit(
  dataset,
  epochs = epochs,
  callbacks = callback_gan_monitor(num_img = 10, latent_dim = latent_dim)
)

```

Создаем экземпляр модели GAN

Вы начнете получать интересные результаты после эпохи 20

В ходе обучения можно заметить, что потери сопоставительной сети значительно возрастают, а потери дискриминатора стремятся к нулю. Иными словами, дискриминатор может слишком быстро научиться побеждать генератор. В этом случае попробуйте уменьшить скорость обучения дискриминатора и увеличить его коэффициент прорезивания. На рис. 12.22 показано, какие изображения наша GAN способна генерировать после 30 эпох обучения.



Рис. 12.22 Примеры сгенерированных изображений после 30-й эпохи обучения

### 12.5.7 Подведение итогов

- Генеративно-сопоставительная сеть состоит из двух сетей: генератора и дискриминатора. Дискриминатор обучается отличать изображения, созданные генератором, от настоящих, имеющих в обучающем наборе, а генератор обучается обманывать дискриминатор. Примечательно, что генератор вообще не видит изобра-

жений из обучающего набора; вся информация, которую он имеет, поступает из дискриминатора.

- Генеративно-сопоставительные 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-сопоставительные 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 видов практического применения, таких как редактирование изображений с использованием концептуальных векторов.

Несколько методов, которые мы рассмотрели в этой главе, охватывают только основы этой стремительно развивающейся области. Вам еще многое предстоит узнать – генеративное глубокое обучение достойно отдельной книги.

## *Краткие итоги главы*

- Вы можете использовать модель преобразования последовательностей для поэлементного создания данных новой последовательности. Этот прием применим к генерации текста, а также к генерации музыки по нотам или любому другому типу данных временных рядов.
- DeepDream работает путем максимизации активаций слоя сверточной сети через градиентное восхождение во входном пространстве.
- В алгоритме переноса стиля содержание одного изображения и стиль другого объединяются посредством градиентного спуска для создания изображения, сохраняющего высокоуровневые абстрактные признаки источника содержания и локальные признаки источника стиля.
- Вариационные автокодировщики и генеративно-сопоставительные сети – это модели, которые изучают скрытое пространство изображений, а затем могут создавать совершенно новые изображения путем выбора точки данных из скрытого пространства. Концептуальные векторы из скрытого пространства можно использовать даже для редактирования изображений.

# 13

## Глубокое обучение в реальной жизни

---

### *Эта глава охватывает следующие темы:*

- настройка гиперпараметров;
- ансамблирование моделей;
- обучение со смешанной точностью;
- обучение моделей Keras на нескольких GPU или TPU.

К этому моменту вы далеко продвинулись в овладении приемами построения моделей глубокого обучения с помощью фреймворка Keras. Теперь вы умеете обучать модели классификации изображений, сегментации изображений, классификации или регрессии векторных данных, прогнозирования временных рядов, классификации текста, преобразования последовательностей и даже модели для генерации текста и изображений. Фактически вы овладели всеми базовыми навыками.

Однако до сих пор все ваши модели действовали в скромном масштабе – на небольших наборах данных, с одним графическим процессором – и, как правило, не достигали максимума точности и быстродействия на каждом наборе данных, который мы рассматривали. В конце концов, эта книга является только введением в область глубокого обучения. Если вы хотите выйти в реальный мир и добиться значимых результатов в решении совершенно новых задач, вам все равно придется перешагнуть через небольшую пропасть.

Эта предпоследняя глава посвящена преодолению разрыва между базовыми знаниями и практикой и содержит отсылки к передовым технологиям, без которых вам не обойтись, если вы хотите превратиться из новичка, изучающего машинное обучение, в продвинутого специалиста. Мы рассмотрим основные методы систематического улучшения качества модели: настройку гиперпараметров и ансамблирование моделей. Затем вы узнаете, как можно ускорить и расширить масштаб обучения моделей с использованием нескольких GPU и TPU, смешанной точности и удаленных вычислительных ресурсов в облаке.

Мы также воспользуемся этой главой, чтобы показать, как получить прямой доступ к пакетам Python, даже если нет подходящей оболочки R. Этот навык пригодится, когда вы продолжите работу с глубоким обучением. Вам не нужны обширные знания языка Python, чтобы использовать пакеты Python из R, но если вы когда-нибудь читали документацию по Python и задавались вопросами вроде «Что означают символы подчеркивания?», обратитесь к приложению, которое поможет пользователям R быстро освоиться с языком Python.

## 13.1 Получение максимальной отдачи от ваших моделей

Использование простых стандартных архитектур – неплохой подход, когда просто-напросто нужно, чтобы все работало. В этом разделе мы сделаем шаг от «просто работающих» к «продвинутым моделям, побеждающим в состязаниях по машинному обучению» и рассмотрим некоторые приемы создания 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 функцию активации `relu` или какую-то другую? Использовать ли `layer_batch_normalization()` после данного слоя? Какой коэффициент прореживания выбрать? И т. д. 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. Что-

бы дойти до самого предела возможностей в решении какой-либо задачи, нельзя довольствоваться произвольным выбором, сделанным по ошибке. Ваши первоначальные решения часто будут неоптимальными, даже если вы обладаете хорошей интуицией. Вы можете менять свой выбор, выполняя настройки вручную и повторно обучая модель, – именно этим большую часть времени занимаются инженеры и исследователи машинного обучения. Однако перебор разных параметров не должен быть вашей основной работой – это дело лучше доверить машине.

Другими словами, вам нужно автоматически, систематически и принципиально исследовать пространство возможных решений. Вам нужно эмпирически исследовать пространство архитектур и найти ту из них, которая сможет обеспечить лучшее качество. Именно эту задачу решает автоматическая оптимизация гиперпараметров: это огромная и очень важная область исследований.

Вот как выглядит типичный процесс оптимизации гиперпараметров:

- 1 выбрать набор гиперпараметров (автоматически);
- 2 создать соответствующую модель;
- 3 обучить ее на обучающих данных и оценить качество на проверочных данных;
- 4 выбрать следующий набор гиперпараметров (автоматически);
- 5 повторить;
- 6 получить окончательную оценку качества на контрольных данных.

Ключевое 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, 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 метод градиентного спуска;
- вычисление сигнала обратной связи (действительно ли данный набор гиперпараметров ведет к увеличению качества модели для данной задачи?) может обходиться очень дорого: для этого нужно создать и обучить новую модель с нуля;

- сигнал обратной связи может быть зашумлен: если обучающий прогон работает на 0,2 % лучше, то причина в лучшей конфигурации модели или вам просто повезло с начальными значениями веса?

К счастью, существует инструмент, упрощающий настройку гиперпараметров, под названием KerasTuner. Давайте проверим его в действии.

## ИСПОЛЬЗОВАНИЕ KERASTUNER

Начнем с установки пакета KerasTuner для Python:

```
reticulate::py_install("keras-tuner", pip = TRUE)
```

KerasTuner позволяет заменять жестко заданные значения гиперпараметров, например `units = 32`, диапазоном возможных вариантов, например `Int(name = "units", min_value = 16, max_value = 64, step = 16)`. Этот набор вариантов выбора в данной модели называется *пространством поиска* процесса настройки гиперпараметров. Чтобы указать пространство поиска, определите функцию построения модели (листинг 13.1). Она принимает аргумент `hp`, из которого извлекает диапазоны гиперпараметров, и возвращает скомпилированную модель Keras.

### Листинг 13.1 Функция построения модели KerasTuner

```

                                Извлечение значений гиперпараметров из объекта hp.
                                Затем эти значения становятся обычными константами R
build_model <- function(hp, num_classes = 10) {

  units <- hp$Int(name = "units",      ←
                 min_value = 16L, max_value = 64L, step = 16L)
  model <- keras_model_sequential() %>%
    layer_dense(units, activation = "relu") %>%
    layer_dense(num_classes, activation = "softmax")

  optimizer <- hp$Choice(name = "optimizer",
                        values = c("rmsprop", "adam"))

  model %>% compile(optimizer = optimizer,
                  loss = "sparse_categorical_crossentropy",
                  metrics = c("accuracy"))

  model ←
}

```

Функция возвращает скомпилированную модель

Доступны различные типы гиперпараметров:  
Int, Float, Boolean, Choice

Если вы хотите использовать более модульный и настраиваемый подход к построению модели, вы также можете создать подкласс класса `HyperModel` и определить метод `build`, как показано в листинге 13.2.

Листинг 13.2 Использование класса `HyperModel` в `KerasTuner`

```

kt <- reticulate::import("kerastuner")
SimpleMLP(kt$HyperModel) %py_class% {
  `__init__` <- function(self, num_classes) {
    self$num_classes <- num_classes
  }

  build <- function(self, hp) {
    build_model(hp, self$num_classes)
  }
}

hypermodel <- SimpleMLP(num_classes = 10)

```

При объектно-ориентированном подходе мы можем настроить константы модели, такие как `num_classes`, в качестве аргументов конструктора

Метод `build()` идентичен нашей предыдущей автономной функции `build_model()`, за исключением того, что теперь он вызывается методом подкласса `kt$HyperModel`

### Определение пользовательских классов Python в коде R с помощью `%py_class%`

Оператор `%py_class%` можно использовать для определения пользовательских классов Python в коде R. Он отражает синтаксис Python для определения классов Python и позволяет почти механически переводить Python в R. Это особенно полезно при использовании API Python, которые разработаны на основе подклассов, например `kt$HyperModel`. Эквивалентное определение `SimpleMLP` в Python (которое представлено в документации Python для `KerasTuner`) будет выглядеть так:

```

import kerastuner as kt

class SimpleMLP(kt.HyperModel):
    def __init__(self, num_classes):
        self.num_classes = num_classes

    def build(self, hp):
        return build_model(hp, self.num_classes)

hypermodel = SimpleMLP(num_classes=10)

```

Воспользуйтесь встроенной справкой `?%py_class%` в R для получения дополнительной информации и примеров.

Далее нужно определить *настройщик* (tuner). В общих чертах настройщик можно представить как цикл `for`, который будет неоднократно выполнять следующие действия:

- выбирать набор значений гиперпараметров;
- вызывать функцию построения модели с этими значениями, чтобы создать модель;
- обучать модель и записывать ее показатели.

KerasTuner имеет несколько встроенных настройщиков – RandomSearch, BayesianOptimization и Hyperband. Попробуем применить настройщик BayesianOptimization, делающий «умный» прогноз новых значений гиперпараметров, исходя из результатов испытаний предыдущих вариантов:

```
tuner <- kt$BayesianOptimization(
  build_model,
  objective = "val_accuracy",
  max_trials = 100L,
  executions_per_trial = 2L,
  directory = "mnist_kt_test",
  overwrite = TRUE
)
```

Указываем метрику, которую настройщик будет стремиться оптимизировать. Всегда указывайте метрики проверки, потому что цель процесса поиска – найти наиболее обобщающие модели

Указываем функцию построения модели (или экземпляр гипермодели)

Максимальное количество различных конфигураций модели («испытаний»), которые нужно попробовать перед завершением поиска

Место для сохранения журнала поиска

Чтобы уменьшить разброс значений метрик, вы можете обучать одну и ту же модель несколько раз и усреднять результаты. `executes_per_trial` – сколько тренировочных раундов (выполнений) нужно запустить для каждой конфигурации модели (испытания)

Следует ли перезаписывать данные в каталоге, чтобы начать новый поиск. Установите значение `TRUE`, если вы изменили функцию построения модели, или `FALSE`, чтобы возобновить ранее начатый поиск с той же функцией построения модели

Вы можете отобразить обзор пространства поиска 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}
```

### Максимизация и минимизация целевой метрики

В случае встроенных метрик (таких как точность в нашем случае) направление оптимизации KerasTuner определяет автоматически (точность должна быть максимальной, но потери должны быть минимизированы). Однако для пользовательской метрики вы должны указать направление самостоятельно, например:

```

objective <- kt$Objective(
  name = "val_accuracy",
  direction = "max"
)
tuner <- kt$BayesianOptimization(
  build_model,
  objective = objective,
  ...
)

```

Имя метрики, указанное в журнале эпохи

Нужное направление метрики: min или max

Наконец, давайте запустим поиск гиперпараметров. Не забудьте передать проверочные данные и убедитесь, что ваш контрольный набор не используется в качестве проверочного, иначе вы быстро начнете подгонять гиперпараметры под контрольный набор и больше не сможете доверять метрикам при прогоне модели на контрольных данных:

```

c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()
x_train %<-% { array_reshape(., c(-1, 28 * 28)) / 255 }
x_test %<-% { array_reshape(., c(-1, 28 * 28)) / 255 }
x_train_full <- x_train
y_train_full <- y_train
num_val_samples <- 10000
c(x_train, x_val) %<-%
  list(x_train[seq(num_val_samples), ],
        x_train[-seq(num_val_samples), ])
c(y_train, y_val) %<-%
  list(y_train[seq(num_val_samples)],
        y_train[-seq(num_val_samples)])

```

Резервируем на будущее

Отделяем проверочный набор

```

callbacks <- c(
  callback_early_stopping(monitor = "val_loss",
                           patience = 5)
)
tuner$search(
  x_train, y_train,
  batch_size = 128L,
  epochs = 100L,
  validation_data = list(x_val, y_val),
  callbacks = callbacks,
  verbose = 2L
)

```

Функция принимает те же аргументы, что и fit() (она просто передает их в fit() для каждой новой модели)

Задайте большое количество эпох (вы не знаете заранее, сколько эпох потребуется каждой модели) и используйте callback\_early\_stopping(), чтобы остановить обучение, когда модель начнет переобучаться

Обязательно передавайте значения типа int там, где их ожидают функции Python, а не значения double

Предыдущий пример будет выполнен всего за несколько минут, потому что мы рассматриваем только несколько возможных вариантов и обучаем модель на наборе MNIST. Однако в реальной жиз-

ни настройщик гиперпараметров может проработать всю ночь или даже несколько дней. Если в процессе поиска произойдет сбой, вы всегда можете перезапустить его – просто передайте настройщику параметр `overwrite = FALSE`, чтобы он мог возобновить работу с последней записи журнала поиска, хранящегося на диске. После завершения поиска вы получите наилучшую конфигурацию гиперпараметров (листинг 13.3), которую можно использовать для создания высококачественных моделей, а затем обучить их заново на прикладных данных.

### Листинг 13.3 Извлечение лучших конфигураций гиперпараметров

```
top_n <- 4L
best_hps <- tuner$get_best_hyperparameters(top_n)
```

Возвращает список объектов `HyperParameter`, которые вы можете передать в функцию построения модели

Обычно при повторном обучении этих моделей проверочные данные используют как часть обучающих данных, потому что вы больше не будете вносить какие-либо изменения гиперпараметров и, следовательно, больше не будете оценивать метрики модели на проверочных данных. В нашем примере мы будем обучать окончательные модели на всей совокупности исходных обучающих данных MNIST, не отделяя проверочный набор.

Но прежде чем приступить к обучению на полном наборе данных, нам нужно выбрать последний параметр: оптимальное количество эпох обучения. Как правило, новые модели обучают дольше, чем во время поиска гиперпараметров: использование агрессивного значения `patience` в `callback_early_stopping()` экономит время поиска, но может привести к недообучению моделей. Просто используйте проверочный набор, чтобы найти лучшую эпоху:

```
get_best_epoch <- function(hp) {
  model <- build_model(hp)

  callbacks <- c(
    callback_early_stopping(monitor = "val_loss", mode = "min",
                             patience = 10))
  history <- model %>% fit(
    x_train, y_train,
    validation_data = list(x_val, y_val),
    epochs = 100,
    batch_size = 128,
    callbacks = callbacks
  )

  best_epoch <- which.min(history$metrics$val_loss)
  print(glue::glue("Best epoch: {best_epoch}"))
  invisible(best_epoch)
}
```

Обратите внимание на большое значение параметра `patience`

Наконец, имеет смысл слегка увеличить количество эпох при обучении на полном наборе, потому что с увеличением объема данных снижается риск переобучения. В нашем примере количество эпох увеличивается на 20 %:

```
get_best_trained_model <- function(hp) {  
  best_epoch <- get_best_epoch(hp)  
  model <- build_model(hp)  
  model %>% fit(  
    x_train_full,  
    y_train_full,  
    batch_size = 128,  
    epochs = round(best_epoch * 1.2)  
  )  
  model  
}  
  
best_models <- best_hps %>%  
  lapply(get_best_trained_model)
```

Если вас не беспокоит небольшая недообученность модели, вы можете пойти по более короткому пути: просто используйте настройщик, чтобы перезагрузить самые эффективные модели с лучшими весами, сохраненными во время поиска гиперпараметров, без повторного обучения новых моделей с нуля:

```
best_models <- tuner$get_best_models(top_n)
```

Одной из важных проблем, которую нужно учитывать при автоматической оптимизации гиперпараметров, является переобучение на проверочном наборе. Поскольку вы обновляете гиперпараметры на основе сигнала, который вычисляется с использованием результатов проверки, тем самым вы обучаете их на проверочных данных, и при достаточно долгом поиске настройщик подгоняет гиперпараметры к проверочным данным. Не забывайте об этом!

## НАУЧИТЕСЬ ВЫБИРАТЬ ПРАВИЛЬНОЕ ПРОСТРАНСТВО ПОИСКА

В целом оптимизация гиперпараметров – это мощный прием, который абсолютно необходимо применять для создания современных моделей или для победы в соревнованиях по машинному обучению. Подумайте о том, что когда-то люди вручную создавали признаки, которые использовались в неглубоких моделях машинного обучения. Это было очень неоптимально. Теперь глубокое обучение автоматизирует задачу поиска иерархических признаков – они изучаются с использованием сигнала обратной связи, а не настраиваются вручную, и это правильно. Точно так же вам не следует создавать точную архитектуру модели вручную; вы должны оптимизировать ее, применяя автоматический поиск гиперпараметров.

Однако настройка гиперпараметров не отменяет необходимость знать методы построения передовых архитектур моделей. Про-

странства поиска растут комбинаторно с ростом количества вариантов, поэтому будет слишком накладно заставлять настройщик подбирать все настраиваемые параметры модели. Вы должны обдуманно подходить к выбору правильного пространства поиска гиперпараметров. Настройка гиперпараметров – это автоматизация, а не волшебство: вы используете настройщик для автоматизации экспериментов, которые в противном случае вы бы проводили вручную, но вам все равно нужно вручную выбирать конфигурации экспериментов, которые потенциально способны дать хорошие показатели.

Хорошая новость заключается в том, что благодаря настройке гиперпараметров вы переходите от микрорешений (какое количество модулей выбрать для этого уровня?) к решениям более высокого уровня (следует ли использовать остаточные связи в этой модели?). В то время как микрорешения специфичны для определенной модели и определенного набора данных, решения более высокого уровня лучше обобщаются для разных задач и наборов данных. Например, практически любую проблему классификации изображений можно решить с помощью одного и того же шаблона пространства поиска.

Следуя этой логике, KerasTuner пытается предложить разработчикам моделей *предварительно созданные области поиска*, которые адаптированы к широким категориям проблем, таким как классификация изображений. Достаточно предоставить данные, запустить поиск гиперпараметров, и вы получите довольно хорошую модель. Например, попробуйте использовать шаблоны гипермоделей `kt$applications$HyperXception` и `kt$applications$HyperResNet`, которые являются настраиваемыми версиями моделей Keras Applications.

## Будущее настройки гиперпараметров: автоматизированное машинное обучение

В настоящее время основная часть работы инженера по глубокому обучению состоит из обработки данных с помощью скриптов R, а затем детальной настройки архитектуры и гиперпараметров глубокой сети, чтобы получить работающую модель – или даже оригинальную передовую модель, если разработчик настолько амбициозен. Излишне говорить, что это далеко не оптимальный рабочий процесс. Здесь на помощь вновь приходит автоматизация, и она не ограничивается простой настройкой гиперпараметров.

Поиск по набору возможных скоростей обучения или возможных размеров слоя – это только первый шаг. Мы достойны большего, не так ли? Хотелось бы иметь возможность генерировать *архитектуру модели* с нуля и с минимальным количеством ограничений, например с помощью обучения с подкреплением или генетических алгоритмов. В будущем целые сквозные конвейеры машинного обучения будут генерироваться автоматически, а не создаваться инженерами-ремесленниками вручную. Эта технология будущего

называется *автоматическим машинным обучением*, или AutoML. Вы уже можете использовать такие библиотеки, как AutoKeras (<https://github.com/keras-team/autokeras>), для решения основных задач машинного обучения с минимальными усилиями с вашей стороны.

Сегодня AutoML находится в зачаточном состоянии и не может решать большие задачи. Но когда технология AutoML станет достаточно зрелой для широкого внедрения, рабочие места инженеров по машинному обучению не исчезнут – наоборот, инженеры перейдут на ступень выше в цепочке создания стоимости продукта. У них появится возможность прилагать гораздо больше усилий к обработке данных, созданию сложных функций потерь, которые лучше отражают бизнес-цели предприятия, а также к пониманию того, как их модели влияют на цифровые экосистемы, в которых они развернуты (например, пользователи, которые используют прогнозы модели и генерируют новые обучающие данные). Это задачи, которые в настоящее время могут осилить только крупнейшие компании.

Всегда охватывайте взором общую картину, старайтесь понять основы и помните, что скучная однообразная деятельность в конечном итоге будет автоматизирована. Рассматривайте это как подарок – повышение производительности ваших рабочих процессов, – а не как угрозу вашей значимости. Ваша работа не должна сводиться к бесконечному вращению ручек настройки.

### 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, 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 части, туловища или ноги. 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 (определяемых уникальной архитектурой модели и случайными значениями весов, полученных в момент инициали-

зации). 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 полнотой истины, 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)

```

Четыре различные 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 случайного поиска или простой оптимизации, такой как метод Нелдера–Мида:

```

preds_a <- model_a %>% predict(x_val)
preds_b <- model_b %>% predict(x_val)
preds_c <- model_c %>% predict(x_val)
preds_d <- model_d %>% predict(x_val)
final_preds <-
  (0.5 * preds_a) + (0.25 * preds_b) +
  (0.1 * preds_c) + (0.15 * preds_d)

```

Эти веса (0.5, 0.25, 0.1, 0.15),  
как предполагается, получены  
эмпирическим путем

Существует много возможных вариантов: вы можете вычислить среднее экспоненциальное прогнозов. 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. Сила в разнообразии. Если бы все незрячие мудрецы ощупали только хобот слона, они согласились бы, что слоны похожи на змей, и навсегда 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, – строить ансамбль из экземпляров одной и той же сети, обученной 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 заняли четвертое место среди решений задачи обнаружения бозона Хиггса на сайте Kaggle ([www.kaggle.com/c/higgs-boson](http://www.kaggle.com/c/higgs-boson)), используя 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 улучшает ансамбль в целом, потому что сильно отличается от всех других моделей: она извлекала 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-участники.

## 13.2 Масштабируемое обучение моделей

Вспомним «цикл прогресса», с которым вы познакомились в главе 7: качество ваших идей зависит от того, сколько циклов уточнения они прошли (рис. 13.1 для удобства повторяет рис. 7.6). Скорость, с которой вы можете повторять цикл, зависит от того, насколько быстро вы можете подготовить эксперимент, как быстро вы можете его провести и, наконец, насколько хорошо вы сможете проанализировать полученные данные.

По мере развития навыков работы с API Keras скорость, с которой вы пишете код для экспериментов по глубокому обучению, перестанет быть узким местом этого цикла прогресса. Следующим узким местом станет скорость, с которой вы можете обучать свои модели. Благодаря доступности готовой инфраструктуры глубокого обуче-

ния вы можете получить результаты через 10–15 минут, а значит, сможете выполнять десятки итераций каждый день. Более быстрое обучение напрямую улучшает качество ваших решений.

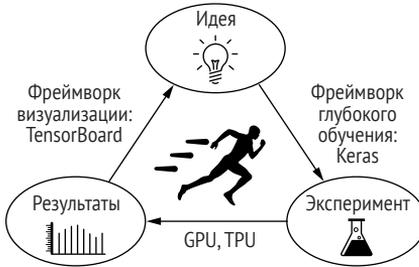


Рис. 13.1 Цикл прогресса

В этом разделе вы узнаете о трех способах ускорить обучение моделей:

- обучение со смешанной точностью, которое можно использовать даже с одним графическим процессором;
- обучение на нескольких графических процессорах;
- обучение на ТПУ.

Разберем эти способы подробнее.

### 13.2.1 Ускорение обучения на GPU со смешанной точностью

Представьте, что существует простой и доступный способ ускорить обучение практически любой модели в три раза, причем практически бесплатно. Звучит слишком хорошо, чтобы быть правдой, и тем не менее такой способ существует – это обучение со смешанной точностью. Чтобы понять, как он работает, сначала нужно разобраться с понятием точности в информатике.

#### Понятие точности чисел с плавающей запятой

Точность для чисел – то же, что разрешение для изображений. Поскольку компьютеры могут обрабатывать только единицы и нули, любое число, которое видит компьютер, должно быть представлено в виде двоичной строки. Например, вы наверняка знакомы с целочисленным форматом `uint8`, который представляет целые числа, закодированные восемью битами: `00000000` представляет 0, а `11111111` представляет 255. Чтобы представить целые числа больше 255, вам нужно больше битов – восьми недостаточно. Для двоичной записи целых чисел часто используют 32 бита, с помощью которых вы можете представлять целые числа со знаком в диапазоне от  $-2\,147\,483\,648$  до  $2\,147\,483\,647$ .

Сказанное выше относится и к числам с плавающей запятой. В математике действительные числа образуют непрерывную ось: между любыми двумя числами находится бесконечное количество

точек. В информатике это не так: например, между числами 3 и 4 существует конечное число промежуточных точек. Но сколько именно? Это зависит от доступной вам *точности* – от количества битов, которые вы используете для хранения числа. Вы можете увеличить точность только до определенного предела. Существуют три уровня точности, которые обычно используют в информатике:

- *половинная*, или float16 (16 бит);
- *одинарная*, или float32 (32 бита);
- *двойная*, или float64 (64 бита).

Разрешение, обеспечиваемое числами с плавающей запятой, можно рассматривать как наименьшее расстояние между двумя произвольными числами, которое вы сможете безопасно обработать. В случае одинарной точности это приблизительно  $10^{-7}$ . Двойная точность обеспечивает разрешение около  $10^{-16}$ , а половинная точность – всего лишь  $10^{-3}$ .

Почти все модели в этой книге используют числа одинарной точности – они сохраняют свое состояние в виде весовых переменных типа float32 и выполняют вычисления над входными данными типа float32. Этой точности достаточно, чтобы выполнить прямой и обратный проходы модели без потери какой-либо информации, особенно когда речь идет о небольших обновлениях градиента (напомним, что типичная скорость обучения составляет  $10^{-3}$ , и довольно часто можно увидеть обновления веса порядка  $10^{-6}$ ).

Разумеется, можно использовать двойную точность, хотя это было бы расточительно – такие операции, как матричное умножение или сложение с двойной точностью, обходятся намного дороже в вычислительном отношении, поэтому вам придется удвоить вычислительную нагрузку на оборудование без каких-либо явных преимуществ. С другой стороны, в глубоком обучении невозможно нормально использовать формат половинной точности float16 для весов и вычислений; процесс градиентного спуска не будет проходить гладко, потому что половинная точность не позволяет представить небольшие обновления градиента.

Однако вы можете использовать гибридный подход: в этом и заключается смысл *смешанной точности*. Идея состоит в том, чтобы использовать 16-битные вычисления там, где не нужна высокая точность, и работать с 32-битными представлениями в остальных случаях, чтобы поддерживать числовую стабильность. Современные GPU и TPU оснащены специальными модулями, которые выполняют 16-разрядные операции намного быстрее и занимают меньше памяти по сравнению с эквивалентными 32-разрядными операциями. Используя операции с половинной точностью, вы можете значительно ускорить обучение модели на GPU и TPU. В то же время, сохраняя одинарную точность для критических частей модели, вы можете избежать существенного снижения ее качества.

Выгода от использования смешанной точности весьма заметна – на современных графических процессорах NVIDIA смешанная точ-

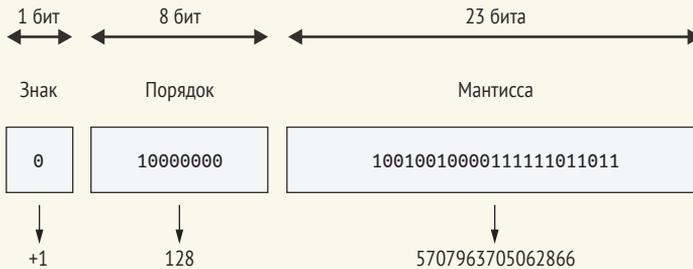
ность может ускорить обучение до 3 раз. Смешанная точность также полезна при обучении на TPU (тема, к которой мы скоро вернемся), где она может ускорить обучение до 60 %.

### Примечание о кодировании чисел с плавающей запятой

Противоречащий здравому смыслу факт о числах с плавающей запятой заключается в том, что представляемые числа не распределены равномерно. Большие числа имеют меньшую точность: между  $2^N$  и  $2^{(N + 1)}$  имеется такое же количество представляемых значений, как и между 1 и 2, для любого  $N$ . Дело в том, что числа с плавающей запятой состоят из трех частей: *знак* (sign), *порядок* (exponent) и *мантисса* (mantissa), записанных в следующем виде:

$$\langle \text{sign} \rangle * (2 ^ { (\langle \text{exponent} \rangle - 127) }) * 1. \langle \text{mantissa} \rangle$$

Например, на следующем рисунке показано, как можно закодировать ближайшее значение float32, близкое к  $\pi$ .



$$\text{value} = +1 * (2 ^ { (128 - 127) }) * 1.5707963705062866$$

$$\text{value} = 3.1415927410125732$$

**Число  $\pi$ , закодированное с одинарной точностью с помощью бита знака, целочисленного порядка и целочисленной мантиссы**

По этой причине числовая ошибка, возникающая при преобразовании числа в его представление с плавающей запятой, может сильно различаться в зависимости от рассматриваемого точного значения, и эта ошибка имеет тенденцию увеличиваться для чисел с большим абсолютным значением.

### Остерегайтесь использовать значения dtype по умолчанию

Одинарная точность – это тип чисел с плавающей запятой по умолчанию в Keras и TensorFlow: создаваемый вами тензор или переменная будут представлены в float32, если вы не укажете иное. Однако для массивов R тип по умолчанию – float64!

Преобразование массива R в тензор TensorFlow даст тензор типа float64, хотя такая точность может быть вам не нужна:

```
r_array <- base::array(0, dim = c(2, 2))
tf_tensor <- tensorflow::as_tensor(r_array)
tf_tensor$dtype
```

```
tf.float64
```

Не забудьте указать типы данных при преобразовании массивов R:

```
r_array <- base::array(0, dim = c(2, 2))
tf_tensor <- tensorflow::as_tensor(r_array, dtype = "float32")
tf_tensor$dtype
```

↑ Укажите тип  
в явном виде

```
tf.float32
```

Учтите, что когда вы вызываете метод Keras `fit()` с массивами R, он по умолчанию автоматически приводит их к `k_floatx()` – `float32`.

## ОБУЧЕНИЕ СО СМЕШАННОЙ ТОЧНОСТЬЮ НА ПРАКТИКЕ

При обучении модели на графическом процессоре вы можете включить смешанную точность следующим образом:

```
keras::keras$mixed_precision$set_global_policy("mixed_float16")
```

↑ `keras::keras` – это модуль Python,  
импортированный при помощи `reticulate`

Как правило, большая часть прямого прохода модели будет выполняться в `float16` (за исключением численно неустойчивых операций, таких как `softmax`), тогда как веса модели будут храниться и обновляться в `float32`.

Слои Keras имеют свойства `variable_dtype` и `compute_dtype`. По умолчанию оба они содержат значение `float32`. Когда вы включаете режим смешанной точности, значение `compute_dtype` большинства слоев меняется на `float16`. Следовательно, эти слои будут приводить свои входные данные к `float16` и выполнять вычисления в `float16` (используя копии весов с половинной точностью). Однако, поскольку их свойство `variable_dtype` по-прежнему имеет значение `float32`, их веса смогут получать от оптимизатора обновления с одинарной точностью типа `float32`, а не обновления с половинной точностью.

Обратите внимание, что при использовании половинной точности некоторые операции (в частности, `softmax` и кросс-энтропия) могут быть численно неустойчивыми. Если вам нужно отказаться от смешанной точности для определенного слоя, просто передайте аргумент `dtype="float32"` в конструктор этого слоя.

### 13.2.2 Обучение модели на нескольких GPU

Хотя графические процессоры с каждым годом становятся все мощнее, модели глубокого обучения развиваются еще быстрее

и требуют все больше вычислительных ресурсов. Обучение модели на одном графическом процессоре сильно ограничивает скорость, с которой вы можете выполнять свою работу. Что делать? Вы можете построить систему с несколькими GPU и применить *распределенное обучение*.

Существует два способа распределения вычислений между несколькими устройствами: *параллелизм на уровне данных* и *параллелизм на уровне модели*.

При параллелизме на уровне данных одну модель реплицируют на несколько вычислительных устройств. Каждая из реплик модели обрабатывает свой пакет данных, а затем диспетчер параллелизации объединяет их результаты.

При параллелизме на уровне модели разные части одной модели работают на разных устройствах, одновременно обрабатывая один и тот же пакет данных. Этот прием лучше всего работает с моделями, имеющими естественную параллельную архитектуру (например, модели с несколькими ветвями). На практике параллелизм на уровне модели используется только для моделей, которые слишком велики, чтобы поместиться на одном устройстве, – это способ обучения очень крупных моделей, а не способ ускорить обучение. В этой книге мы рассмотрим только параллелизм на уровне данных, потому что вы будете использовать его в большинстве случаев в своей практике. Давайте посмотрим, как он работает.

## Доступ к нескольким графическим процессорам

Прежде всего вам нужно иметь доступ к нескольким графическим процессорам. Есть два возможных варианта:

- приобрести от двух до четырех графических процессоров, смонтировать их в одну машину (нужен мощный блок питания) и установить драйверы CUDA, cuDNN и т. д. Для большинства пользователей такой вариант неприемлем из-за высокой стоимости GPU;
- арендовать *виртуальную машину* (virtual machine, VM) с несколькими графическими процессорами в Google Cloud, Azure или AWS (Российским пользователям доступен сервис Yandex Cloud. – *Прим. перев.*). Вы сможете использовать образы VM с предустановленными драйверами и программным обеспечением, при этом затраты на настройку будут минимальными. Вероятно, это лучший выбор для тех, кто не обучает модели круглосуточно и без выходных.

Я не буду подробно рассказывать о том, как развернуть облачные виртуальные машины с несколькими GPU, потому что такие инструкции быстро теряют актуальность, а свежую информацию легко найти в интернете.

## СИНХРОННОЕ ОБУЧЕНИЕ НА НЕСКОЛЬКИХ УСТРОЙСТВАХ

Научитесь выполнять вызов `library(tensorflow)` на машине с несколькими графическими процессорами, и следующим шагом будет обучение распределенной модели. Рассмотрим следующий код:

```

library(tensorflow)
strategy <- tf$distribute$MirroredStrategy()
cat("Number of devices:", strategy$num_replicas_in_sync, "\n")
with(strategy$scope(), {
  model <- get_compiled_model()
})
model %>% fit(
  train_dataset,
  epochs = 100,
  validation_data = val_dataset,
  callbacks = callbacks
)

```

Область действия должна быть открыта для всех операций, создающих переменные. Как правило, это только конструктор модели и метод `compile()`

Создаем объект «стратегии распределения». Очевидным решением является `MirroredStrategy()`

Открываем «область действия стратегии»

Обучаем модель на всех доступных устройствах

В этих строках кода реализована наиболее распространенная стратегия обучения: *синхронное обучение на нескольких устройствах с одним хостом*, также известная в TensorFlow как «стратегия зеркального распределения». «Один хост» означает, что все доступные графические процессоры установлены на одной машине (в отличие от *кластера* из нескольких машин, каждая со своим собственным графическим процессором, взаимодействующих по сети). Термин «синхронное обучение» означает, что состояние реплик модели для каждого графического процессора всегда остается одинаковым – существуют варианты распределенного обучения, в которых это не так. Когда вы открываете область действия `MirroredStrategy()` и создаете в ней свою модель, объект `MirroredStrategy()` создает одну копию модели (реплику) на каждом доступном графическом процессоре. Например, если у вас два графических процессора, то каждый шаг обучения разворачивается в следующем порядке (рис. 13.2):

- 1 пакет данных (называемый *глобальным пакетом*) извлекается из набора данных;
- 2 глобальный пакет разделяется на два *локальных пакета*. Например, если в глобальном пакете 256 образцов, в каждом из двух локальных пакетов будет 128 образцов. Поскольку локальные пакеты должны быть достаточно большими, чтобы имело смысл использовать GPU, размер глобального пакета обычно бывает очень большим;
- 3 каждая из двух реплик модели обрабатывает один локальный пакет независимо на своем собственном GPU: они выполняют

прямой, а затем обратный проход. Каждая реплика выводит «дельту веса», описывающую изменение каждой весовой переменной в модели с учетом градиента предыдущих весов по отношению к потерям модели на локальном пакете;

- 4 дельты весов, полученные из локальных градиентов, объединяются для получения глобальной дельты, которая применяется ко всем репликам. Поскольку это делается в конце каждого шага, реплики всегда остаются синхронизированными: их веса всегда совпадают.

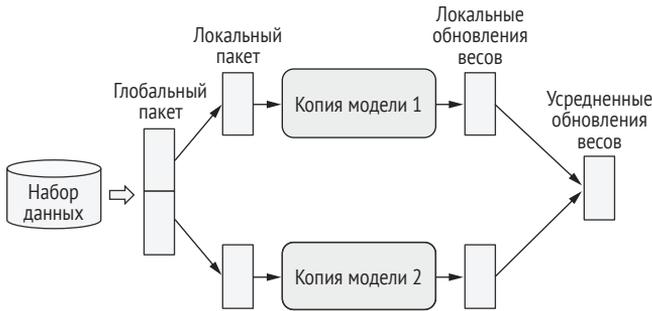


Рис. 13.2 Один шаг обучения `MirroredStrategy`: каждая реплика модели вычисляет локальные обновления весов, которые затем объединяются и используются для обновления состояния всех реплик

Чтобы добиться максимального быстродействия при распределенном обучении, всегда предоставляйте свои данные в виде объекта набора данных TensorFlow. (Передача данных в виде массивов R также работает, потому что они преобразуются в объекты набора данных TensorFlow с помощью `fit()`.) Также необходимо использовать предварительную выборку данных: перед передачей набора данных в `fit()` вызовите функцию `dataset_prefetch(buffer_size)`. Если вы не знаете, какой размер буфера выбрать, попробуйте оставить значение по умолчанию `tf$data$AUTOTUNE`, и фреймворк выберет размер буфера за вас.

В листинге 13.4 показан простой пример построения модели для распределенного обучения.

#### Листинг 13.4 Построение модели в области действия `MirroredStrategy`

```

build_model <- function(input_size) {
  resnet <- application_resnet50(weights = NULL,
                                include_top = FALSE,
                                pooling = "max")

  inputs <- layer_input(c(input_size, 3))

  outputs <- inputs %>%
    resnet_preprocess_input() %>%

```

```

resnet() %>%
  layer_dense(10, activation = "softmax")

model <- keras_model(inputs, outputs)

model %>% compile(
  optimizer = "rmsprop",
  loss = "sparse_categorical_crossentropy",
  metrics = "accuracy"
)

model
}

strategy <- tf$distribute$MirroredStrategy()
cat("Number of replicas:", strategy$num_replicas_in_sync, "\n")

Number of replicas: 2

with(strategy$scope(), {
  model <- build_model(input_size = c(32, 32))
})

```

В данном случае будем обучать модель прямо на массивах R в памяти (которые эффективно преобразуются в набор данных TensorFlow с помощью `fit()`). Воспользуемся набором данных CIFAR10:

```

c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_cifar10()
model %>% fit(x_train, y_train, batch_size = 1024)

```

Обратите внимание, что при обучении с использованием нескольких графических процессоров требуются большие объемы пакетов, чтобы обеспечить эффективное использование устройства



В идеальном мире распределение обучения на  $N$  графических процессоров привело бы к ускорению в  $N$  раз. Однако на практике распределение сопровождается определенными накладными расходами. В частности, объединение разностей весов, поступающих от разных устройств, занимает некоторое время. Реальное ускорение, которое вы получаете, зависит от количества используемых графических процессоров:

- при использовании двух графических процессоров ускорение остается близким к двукратному;
- с четырьмя GPU скорость обучения возрастает примерно в 3,8 раза;
- с восемью GPU вы получите ускорение примерно в 7,3 раза.

Эти рассуждения подразумевают, что вы располагаете достаточно большим глобальным пакетом данных, чтобы каждый графический процессор использовался на полную мощность. Если ваш глобальный пакет данных слишком мал, размера локального пакета будет недостаточно для полной загрузки GPU.

### 13.2.3 Обучение модели на TPU

Помимо обучения моделей на графических процессорах, в мире глубокого обучения набирает популярность перенос рабочих процессов на все более специализированное оборудование, разработанное специально для глубокого обучения. Такие одноцелевые чипы известны как ASIC (application-specific integrated circuits, интегральные схемы для конкретных приложений). Разработкой новых чипов ASIC занимаются различные большие и малые компании, но сегодня наиболее заметные успехи в этом направлении демонстрирует Google со своим чипом Tensor Processing Unit (TPU). Он доступен в сервисах Google Cloud и через Google Colab.

Обучение моделей на TPU требует от разработчика обладания специфическими навыками, но результат того стоит: TPU работают очень, *очень* быстро. Обучение на TPU V2 обычно происходит в 15 раз быстрее, чем обучение на графическом процессоре NVIDIA P100.

Вот несколько советов по использованию TPU. Когда вы используете облачный сервис GPU, ваши модели имеют прямой доступ к графическому процессору без каких-либо специальных действий. В случае использования TPU это не так – есть дополнительный шаг, который нужно сделать, прежде чем вы сможете начать построение модели, – вам нужно подключиться к кластеру TPU. Это делается следующим образом:

```

tpu <- tf$distribute$cluster_resolver$TPUClusterResolver$connect()
cat("Device:", tpu$master(), "\n")
strategy <- tf$distribute$TPUStrategy(tpu) ← Использование TPUStrategy()
with(strategy$scope(), { ... })           ← похоже на tf$distribute$MirroredStrategy()

```

Вам не нужно задумываться о том, что делает этот код, – просто воспринимайте его как небольшое заклинание, которое соединяет вашу среду выполнения с устройством. Сезам, откройся!

Как и в случае обучения с несколькими графическими процессорами, для использования TPU необходимо открыть область действия стратегии распределения – в данном случае область действия `TPUStrategy()`. Стратегия `TPUStrategy()` следует тому же шаблону распределения, что и `MirroredStrategy()`, – модель реплицируется один раз для каждого ядра TPU, а реплики синхронизируются.

У среды выполнения TPU есть одна любопытная особенность: это система с двумя виртуальными машинами, а это означает, что виртуальная машина, на которой размещена среда выполнения в вашем ноутбуке, не является той же виртуальной машиной, в которой работает TPU. Из-за этого вы не сможете использовать для обучения данные, хранящиеся на локальном диске (то есть на диске, связанном с виртуальной машиной, на которой размещен экземпляр). Среда выполнения TPU не может читать их оттуда. У вас есть два способа передачи данных:

- обучение на данных, которые размещены в памяти виртуальной машины (не на диске). Если ваши данные находятся в массиве R, значит, они уже размещены в памяти;
- разместить данные в корзине Google Cloud Storage (GCS) и создать набор данных, который будет считывать данные непосредственно из корзины без загрузки на локальную машину. Среда выполнения TPU может считывать данные из GCS. Это единственный вариант для наборов данных, которые слишком велики, чтобы полностью храниться в памяти.

Нужно заметить, что первая эпоха обучения на TPU начинается не сразу. Дело в том, что ваша модель предварительно компилируется в форму, пригодную для выполнения на TPU. После компиляции само обучение происходит молниеносно.

### Избегайте узких мест ввода-вывода

Поскольку TPU очень быстро обрабатывают пакеты данных, недостаточная скорость получения данных из GCS может легко стать узким местом всей системы:

- если ваш набор данных не очень велик, постарайтесь разместить его в памяти виртуальной машины. Вы можете сделать это, вызвав `dataset_cache()` для набора данных. В этом случае данные будут считаны из GCS только один раз;
- если ваш набор данных не помещается в памяти, обязательно сохраните его в виде файлов TFRecord – эффективного двоичного формата хранения, который можно загрузить очень быстро. На <https://keras.rstudio.com> вы найдете пример кода, демонстрирующий, как преобразовать ваши данные в файлы TFRecord.

## ИСПОЛЬЗОВАНИЕ ПОШАГОВОГО СЛИЯНИЯ ДЛЯ ОПТИМИЗАЦИИ ПРИМЕНЕНИЯ TPU

Поскольку TPU располагают огромной вычислительной мощностью, необходимо обучать модель на очень больших пакетах данных, чтобы оптимально использовать доступные ресурсы. Даже при обучении небольших моделей требуемый размер пакета может быть чрезвычайно большим – до 10 000 образцов на пакет. При работе с такими огромными пакетами вы должны пропорционально увеличить скорость обучения оптимизатора; вы будете реже обновлять веса модели, но каждое обновление будет более точным (поскольку градиенты вычисляются с использованием большего количества точек данных), поэтому вам следует перемещать веса на большую величину с каждым обновлением.

Однако вы можете использовать простой трюк, чтобы сохранить пакеты разумного размера при сохранении полного использования TPU, – *слияние шагов* (step fusing). Идея состоит в том, чтобы выпол-

нять несколько шагов обучения на каждом этапе выполнения TPU. Фактически TPU будет выполнять больше работы между двумя обращениями к памяти виртуальной машины. Для этого передайте в метод `compile()` аргумент `steps_per_execution` – например, `steps_per_execution = 8` для выполнения восьми шагов обучения во время каждого шага выполнения TPU. При обучении небольших моделей, недостаточно использующих TPU (или GPU), это может привести к значительному ускорению.

## Краткие итоги главы

- Вы можете использовать настройку гиперпараметров и Keras-Tuner, чтобы автоматизировать утомительный поиск наилучшей конфигурации модели. Но помните о переобучении гиперпараметров на проверочном наборе!
- Использование ансамбля разнообразных моделей может значительно повысить качество совокупных прогнозов.
- Использование смешанной точности позволяет значительно ускорить обучение модели на графическом процессоре практически без дополнительных накладных расходов.
- Дальнейшее повышение скорости обучения больших моделей достигается при помощи API `tf$distribute$MirroredStrategy()` за счет параллельного применения нескольких графических процессоров.
- Вы можете обучать модели на TPU Google, используя API `TPUStrategy()`. Если объема обучающих данных или сложности модели недостаточно для полной загрузки TPU, обязательно используйте слияние шагов, передав аргумент `compile(..., steps_per_execution = N)` для полного применения ядер TPU.

# 14

## Заключение

---

### *Эта глава охватывает следующие темы:*

- важные уроки книги;
- ограничения глубокого обучения;
- будущее глубокого обучения, машинного обучения и ИИ;
- ресурсы для дальнейшего обучения и использования в работе.

Вы почти достигли конца книги. Эта последняя глава обобщит и повторит основные понятия, а также расширит ваши горизонты за пределы относительно простых понятий, с которыми вы познакомились. Знакомство с глубоким обучением и ИИ – целое путешествие, и конец нашей 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 с высоты птичьего полета все то, что вы должны вынести из этой книги. Это поможет вам освежить в памяти некоторые понятия, изученные здесь. Затем 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. Это должно заинтересовать тех, кто захочет заняться фундаментальными

ми 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

Этот раздел обобщает ключевые выводы из книги. Если вам требуется быстро освежить в памяти все, что вы изучили здесь, прочитайте эти несколько страниц.

### 14.1.1 Различные подходы к ИИ

Прежде всего глубокое обучение не является синонимом ИИ или даже машинного обучения:

- *искусственный интеллект (ИИ)* – это давно существующая обширная 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, прежде чем стать доминирующей формой ИИ в 2000-х;
- *глубокое обучение* – одна из многих ветвей машинного обучения, где модели представлены длинными цепочками геометрических функций, применяемых друг за другом. Эти операции организованы в модули, которые называются слоями, или уровнями: модели глубокого обучения обычно формируются как стек слоев или, в более общем смысле, граф слоев. Эти слои параметризуются весами, в поиске которых и заключается обучение модели. Знание модели хранится в ее весах, а процесс обучения заключается в поиске «правильных значений» для этих весов – значений, которые минимизируют функцию потерь. Поскольку упомянутая цепочка геометрических преобразований дифференцируема, обновление весов для 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 выделяет глубокое обучение среди других подходов к машинному обучению

Всего за несколько лет глубокое обучение достигло огромного успеха в решении широкого круга задач, которые прежде воспринимались как очень сложные для компьютеров, особенно в области машинного восприятия – извлечения полезной информации из изображений, видео, звуков и многого другого. При наличии достаточного объема обучающих данных (например, обучающих данных, предварительно классифицированных людьми) из сенсорной информации можно извлечь почти все то же, что может извлечь человек. Поэтому иногда говорят, что глубокое обучение *решило проблему восприятия*, хотя это верно только для очень узкого определения термина *восприятие*.

Благодаря беспрецедентным техническим успехам глубокое обучение в одиночку принесло третье и, безусловно, самое долгое *лето ИИ*: период повышенного интереса, инвестиций и шумихи в области ИИ. Эта книга как раз писалась в середине этого лета. Завершится ли этот период в ближайшем будущем и что случится в конце – это тема для дискуссий. Одно можно сказать наверняка: в отличие от других летних периодов ИИ, глубокое обучение принесло огромные выгоды ряду крупных технологических компаний, позволив распознавать человеческую речь, оказывать интеллектуальную помощь, классифицировать изображения с точностью на уровне человека, значительно улучшить машинный перевод и многое другое. Шумиха уляжется, 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 сейчас прогрессирует невероятно быстрыми темпами благодаря все возрастающим инвестициям в ресурсы и людей. С той точки, где я нахожусь, будущее представляется ярким, хотя краткосрочные ожидания кажутся чересчур оптимистичными; развертывание глубокого обучения в полную меру его потенциала займет не меньше десятилетия.

### 14.1.3 Как правильно воспринимать глубокое обучение

Самое удивительное в глубоком обучении – простота реализации. Еще десять лет тому назад никто не предполагал, что мы добьемся таких успехов в решении задач машинного восприятия, используя простые параметрические модели, обучаемые методом градиент-

ного 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*: все – *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*, это совершенно необходимо для обучения весов посредством градиентного спуска. Это означает, что геометрическое преобразование входных данных в выходные должно быть гладким и непрерывным, что является существенным ограничением.

Весь процесс применения сложного геометрического преобразования к входным данным можно представить как человека, пытающегося развернуть смятый комочек бумаги: этот комочек – 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-то и заключается магия 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 идее: *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,

---

<sup>1</sup> Интервью с Ричардом Фейнманом, *The World from Another Point of View*, телевидение Йоркшира, 1972.

однако нетрудно представить применение других структур данных для интеллекта, например графов. Первоначально нейронные сети возникли из идеи использования графов как способа кодирования смысла, поэтому они и получили название *нейронные сети*; окружающую область исследований обычно называли *коннекционизмом* (connectionism). В настоящее время название «нейронные сети» сохраняется исключительно благодаря традиции – это название весьма далеко от истины, потому что они не являются ни нейронными, ни сетями. В частности, нейронные сети едва ли имеют какое-то сходство с мозгом. Более подходящим было бы название *обучаемые многоуровневые представления*, или *обучаемые иерархические представления*, или даже *глубокие дифференцируемые модели* или *последовательные геометрические преобразования*, чтобы подчеркнуть непрерывность манипуляций с геометрическим пространством.

### 14.1.4 Ключевые технологии глубокого обучения

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 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 в применении к хранилищам данных;
- *доступность быстродействующего 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, что окупилось с лихвой;
- *формирование комплексного стека программных технологий, которые сделали эту вычислительную мощь доступной для людей*: языка CUDA, а также фреймворков, таких как TensorFlow, автоматически выполняющих дифференцирование, и Keras, обеспечивающих доступность глубокого обучения для многих.

В будущем глубокое обучение будет использоваться не только специалистами – учеными, аспирантами и инженерами академического профиля, но и любыми разработчиками, как это произошло с веб-технологиями. Всем, кому необходимы интеллектуальные приложения: так же как любой компании в наши дни требуется свой веб-сайт, каждому продукту будет нужно интеллектуально осмысливать данные, генерируемые пользователями. Для приближения этого будущего мы должны создавать инструменты, которые делают глубокое обучение радикально простым в использовании и доступным любому, имеющему базовые навыки программирования. Keras – первый важный шаг в этом направлении.

### 14.1.5 *Обобщенный рабочий процесс машинного обучения*

Хорошо иметь доступ к чрезвычайно мощному инструменту создания моделей, отображающих любое входное пространство в любое целевое пространство, 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, такие как 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;
- 7 разверните окончательную модель в рабочей среде – как веб-API, как часть приложения JavaScript или C++, на встроенном устройстве и т. д. Продолжайте наблюдать за точностью модели на реальных данных и используйте полученные результаты для доработки следующей версии модели.

### 14.1.6 Основные архитектуры сетей

Четыре семейства сетевых архитектур, с которыми вы должны быть знакомы, – это *полносвязные*, *сверточные*, *рекуррентные* и *трансформеры*. Каждый тип модели предназначен для конкретной модальности входных данных. В архитектуре сети закодированы *предположения* о структуре данных: *пространство гипотез*, в котором осуществляется поиск хорошей модели. Соответствие выбранной архитектуры данной задаче полностью зависит от соответствия структуры данных предположениям сетевой архитектуры.

Эти разные типы сетей можно объединять для создания больших мультимодальных сетей, подобно деталям конструктора LEGO. В некотором смысле слои глубокого обучения – это детали LEGO для обработки информации. Ниже приводится краткий обзор соответствий между некоторыми входными модальностями и сетевыми архитектурами:

- *векторные данные* – полносвязные сети (полносвязные уровни);
- *изображения* – двумерные сверточные сети;
- *последовательные данные* – RNN для временных рядов или трансформеры для дискретных последовательностей (например, последовательностей слов). Одномерные сверточные сети также можно использовать для инвариантных к трансляции данных непрерывной последовательности, таких как звуковые волны;
- *видеоданные* – либо трехмерные сверточные сети (если вам нужно выделить эффекты движения), либо комбинация из двухмерной сверточной сети с покадровой обработкой для извлечения признаков, за которой следует модель обработки последовательности;
- *объемные данные* – трехмерные сверточные сети.

Теперь вспомним особенности каждой архитектуры.

## Полносвязные сети

Полносвязная сеть – это стек полносвязных слоев, предназначенных для обработки векторных данных (пакетов векторов). Такие сети не предполагают наличия во входных признаках какой-то определенной структуры: они называются *полносвязными* (densely connected), потому что измерения полносвязного слоя 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, использовавшемся в главе 4. Они также используются для заключительной классификации или регрессии в большинстве сетей. Например, сверточные сети, рассмотренные в главе 8, обычно завершаются одним или двумя полносвязными слоями Dense, как и рекуррентные сети в главе 10.

Помните: для *бинарной классификации* стек слоев должен завершаться полносвязным слоем с единственным измерением, функцией активации sigmoid и функцией потерь binary\_crossentropy. Вашими целями должно быть значение 0 или 1:

```
inputs <- layer_input(shape = c(num_inputs_features))
outputs <- inputs %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(1, activation = "sigmoid")
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 уровней полносвязным уровнем Dense с количеством измерений, равным количеству классов, и функцией активации softmax. Если цели получены прямым унитарным кодированием, используйте функцию потерь categorical\_crossentropy; если они – целые числа, используйте sparse\_categorical\_crossentropy:

```
inputs <- layer_input(shape = c(num_inputs_features))
outputs <- inputs %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(num_classes, activation = "softmax")
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 уровней полносвязным уровнем Dense с количеством измерений, равным количеству классов, функцией активации softmax и функцией потерь binary\_crossentropy. Ваши цели должны быть получены  $k$ -мерным прямым кодированием:

```
inputs <- layer_input(shape = c(num_inputs_features))
outputs <- inputs %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(num_classes, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop", loss = "binary_crossentropy")
```

Чтобы выполнить *регрессию* в направлении вектора непрерывных значений, завершайте стек уровней полносвязным слоем Dense с количеством измерений, равным количеству значений, которые вы пытаетесь предсказать (часто одно, например цену на недвижимость), без функции активации. Для регрессии можно использовать несколько функций потерь, наиболее часто на практике используются mean\_squared\_error (MSE):

```
inputs <- layer_input(shape = c(num_inputs_features))
outputs <- inputs %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(num_values)
model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop", loss = "mse")
```

## СВЕРТОЧНЫЕ СЕТИ

Сверточные уровни выделяют локальные пространственные шаблоны, применяя одни и те же геометрические преобразования к разным участкам в пространстве (*патчам* или *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.

*Сверточные нейронные сети* состоят из стека сверточных слоев и слоев выбора максимальных значений из соседних (max-pooling). Слои выбора дают возможность снижать пространственную размерность данных, что необходимо для сохранения размеров карты признаков в разумных пределах с ростом числа признаков, и позволяют последующим сверточным слоям «увидеть» входное пространство на большем протяжении. Сверточные сети часто заканчиваются слоем Flatten или уровнем глобального выбора, превращающими карту

пространственных признаков в векторы, за которыми следуют полносвязные слои Dense, реализующие классификацию или регрессию.

Вот типичная сеть для классификации изображений (в данном случае категориальная классификация) с использованием слоев SeparableConv2D:

```
inputs <- layer_input(shape = c(height, width, channels))
outputs <- inputs %>%
  layer_separable_conv_2d(32, 3, activation = "relu") %>%
  layer_separable_conv_2d(64, 3, activation = "relu") %>%
  layer_max_pooling_2d(2) %>%
  layer_separable_conv_2d(64, 3, activation = "relu") %>%
  layer_separable_conv_2d(128, 3, activation = "relu") %>%
  layer_max_pooling_2d(2) %>%
  layer_separable_conv_2d(64, 3, activation = "relu") %>%
  layer_separable_conv_2d(128, 3, activation = "relu") %>%
  layer_global_average_pooling_2d() %>%
  layer_dense(32, activation = "relu") %>%
  layer_dense(num_classes, activation = "softmax")
model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop", loss = "categorical_crossentropy")
```

При построении очень глубокой сверточной сети обычно добавляют слои *пакетной нормализации*, а также *остаточные связи* – два архитектурных приема, которые помогают градиентной информации без загущания проходить через сеть.

## РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ

*Рекуррентные нейронные сети* (Recurrent Neural Networks, RNN) обрабатывают входные последовательности по одному временному интервалу за раз, поддерживая *состояние* на всем протяжении (обычно состояние – это вектор или набор векторов: точка в геометрическом пространстве состояний). Как правило, они предпочтительнее одномерных сверточных сетей, когда обрабатываются последовательности, где интересующие шаблоны не инвариантны в отношении смещения по времени (например, временные ряды данных, в которых недавнее прошлое важнее отдаленного).

В Keras доступны три слоя RNN: SimpleRNN, GRU и LSTM. Для большинства практических применений лучше использовать GRU или LSTM. Слой LSTM – более мощный из этих двух, но и более дорогостоящий в вычислительном смысле; GRU можно рассматривать как более простую и недорогую альтернативу.

Чтобы уложить в стек несколько уровней RNN, каждый предыдущий слой перед последним должен возвращать полную последовательность своих выходов (каждый входной временной интервал будет соответствовать выходному); если сверху не накладываются никакие другие слои RNN, тогда сеть возвращает только последний вывод, содержащий информацию обо всей последовательности.

Вот единственный простой слой RNN для бинарной классификации последовательностей векторов:

```
inputs <- layer_input(shape = c(num_timesteps, num_features))
outputs <- inputs %>%
  layer_lstm(32) %>%
  layer_dense(num_classes, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop", loss = "binary_crossentropy")
```

А вот стек из слоев RNN для бинарной классификации последовательностей векторов:

```
inputs <- layer_input(shape = c(num_timesteps, num_features))
outputs <- inputs %>%
  layer_lstm(32, return_sequences = TRUE) %>%
  layer_lstm(32, return_sequences = TRUE) %>%
  layer_lstm(32) %>%
  layer_dense(num_classes, activation = "sigmoid")
model <- keras_model(inputs, outputs)
model %>% compile(optimizer = "rmsprop", loss = "binary_crossentropy")
```

## ТРАНСФОРМЕРЫ

*Трансформер* просматривает набор векторов (например, векторов слов) и использует *нейронное внимание* для преобразования каждого вектора в представление, учитывающее контекст, формируемый другими векторами в наборе. Когда рассматриваемый набор данных представляет собой упорядоченную последовательность, вы также можете использовать *позиционное кодирование* для создания трансформеров, которые учитывают как глобальный контекст, так и порядок слов и способны обрабатывать длинные текстовые абзацы гораздо эффективнее, чем RNN или одномерные сверточные сети.

Трансформеры можно использовать для любых задач обработки наборов или последовательностей, включая классификацию текста, но они особенно хороши при обучении преобразованию последовательностей, например при переводе абзацев текста с одного языка на другой. Модель преобразователя последовательностей состоит из двух частей:

- `TransformerEncoder` – преобразует последовательность входных векторов в контекстно-зависимую последовательность выходных векторов с учетом порядка;
- `TransformerDecoder` – принимает выходные данные `TransformerEncoder`, а также целевую последовательность и предсказывает, что должно быть дальше в целевой последовательности.

При обработке единственной последовательности (или набора) векторов применяется только `TransformerEncoder`.

Ниже приведен код модели-трансформера для сопоставления исходной последовательности с целевой (эта архитектура может ис-

пользоваться, например, для машинного перевода или ответов на вопросы):

```

encoder_inputs <- layer_input(shape = c(sequence_length),
                               dtype = "int64")
encoder_outputs <- encoder_inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  layer_transformer_encoder(embed_dim, dense_dim, num_heads)

decoder <- layer_transformer_decoder(NULL, embed_dim, dense_dim, num_heads)
decoder_inputs <- layer_input(shape = c(NA),
                               dtype = "int64")
decoder_outputs <- decoder_inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  decoder(., encoder_outputs) %>%
  layer_dense(vocab_size, activation = "softmax")

transformer <- keras_model(list(encoder_inputs, decoder_inputs),
                             decoder_outputs)

transformer %>%
  compile(optimizer = "rmsprop", loss = "categorical_crossentropy")

```

Исходная последовательность

Целевая последовательность на текущий момент

Целевая последовательность на один шаг в будущем

А это отдельный `TransformerEncoder` для бинарной классификации целочисленных последовательностей:

```

inputs <- layer_input(shape = c(sequence_length), dtype = "int64")
outputs <- inputs %>%
  layer_positional_embedding(sequence_length, vocab_size, embed_dim) %>%
  layer_transformer_encoder(embed_dim, dense_dim, num_heads) %>%
  layer_global_max_pooling_1d() %>%
  layer_dense(1, activation = "sigmoid")
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 LEGO: слои можно подключать друг к другу для отображения практически чего угодно при наличии подходящего набора обучающих данных и возможности получения отображения с помощью последовательности геометрических преобразований с разумной сложностью. Пространство возможностей бесконечно. В этом разделе демонстрируется несколько примеров, чтобы показать, что глубокое обучение позволяет решать не только задачи классификации и регрессии, которые традиционно были хлебом насущным для машинного обучения.

Я отсортировал предлагаемые мною примеры применения по модальностям входов и выходов. Обратите внимание, что некоторые из примеров выходят за 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. Разделы с 14.2 по 14.4 посвящены тому, 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, что он перестанет пользоваться спросом в будущем году;
- отображение изображения в вектор данных:
  - *помощник доктора* – предсказание наличия опухоли по медицинским фотографиям;
  - *транспорт с автоматическим управлением* – определение угла поворота рулевых колес по кадрам, поступающим с видеокамеры;
  - *настольные игры с ИИ* – предсказание следующего хода игрока по расположению фигур на шахматной доске или камней на доске го;
  - *помощник диетолога* – предсказание калорийности блюда по его изображению;
  - *предсказание возраста* – определение возраста людей по их автопортретам (селфи);
- отображение данных временного ряда 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, что пользователь купит что-то, по временной последовательности взаимодействий его с веб-сайтом;
- отображение текста в текст:
  - *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:
  - *транскрипция текста* – преобразование текстовых элементов изображений в соответствующие текстовые строки;
  - *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*. В следующем разделе вы увидите, чего мы не можем 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

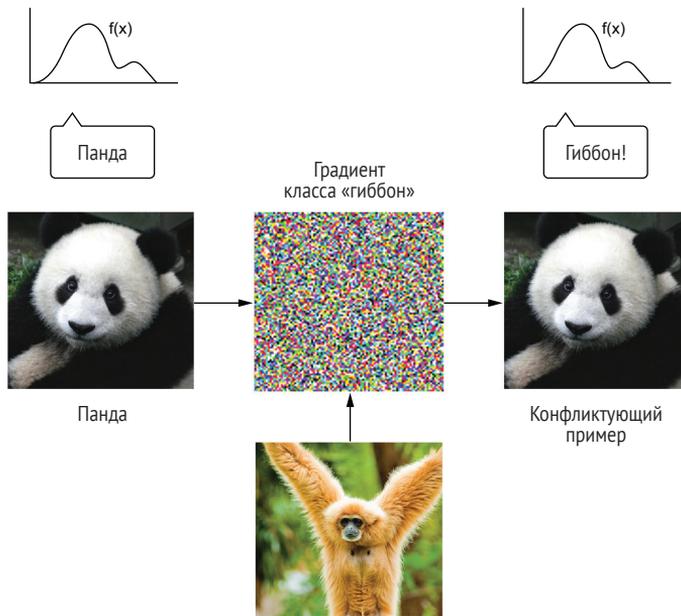
На современном этапе развития ИИ существует реальный риск неверно истолковать, что делают модели глубокого обучения, и переоценить их возможности. Фундаментальной особенностью человека является наша теория разума: наше стремление проецировать намерения, убеждения и знания на окружающий мир. Если нарисовать на скале улыбающийся смайлик, она будет выглядеть «счастливой» – в наших умах. Применительно к глубокому обучению это означает, что когда, например, нам удастся успешно обучить модель, генерирующую подписи к изображениям, мы склонны думать, что модель «понимает», что на них изображено, и генерирует подписи. Но потом мы удивляемся, когда малейшее отступление от изображений, имеющихся в обучающем наборе, заставляет модель генерировать совершенно абсурдные подписи (рис. 14.1).



Мальчик, держащий бесболную биты

Рис. 14.1 Ошибка системы создания подписей к изображению на основе глубокого обучения

Это особенно ярко подчеркивается *примерами с состязательными сетями*, которые передают в сеть глубокого обучения образцы, сконструированные специально, 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, обсуждавшегося 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 глубокого отличия их отображения входов в выходы от человеческого восприятия.



**Рис. 14.2** Незаметные изменения в изображении могут мешать модели правильно его классифицировать

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 потому 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 как тусклое отражение в зеркале (рис. 14.3). Модели, которые вы создаете, будут использовать любую доступную зацепку в данных, чтобы использовать ее для обучения. Например, модели изображений, как правило, больше полагаются на локальные текстуры, чем на глобальное понимание входных изображений – модель, обученная на наборе данных, в котором присутствуют как леопарды, так и диваны, скорее всего, классифицирует диван с «леопардовой» обивкой как настоящего леопарда.

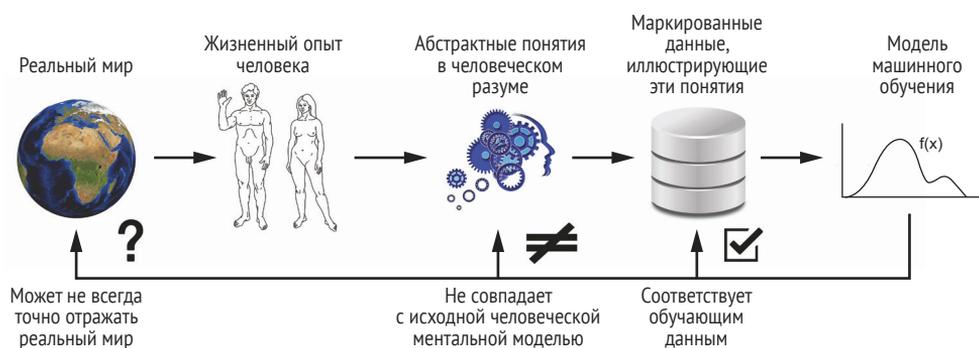


Рис. 14.3 Современные модели машинного обучения похожи на тусклое отражение реальности в зеркале

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. Покажите им что-то, что отклоняется от обучающих данных, и они начнут проявлять абсурдное поведение.

## 14.2.2 Принципиальное различие между автоматом и интеллектом

Существуют фундаментальные различия между цепочкой простых геометрических преобразований входных данных, которые выполняют модели глубокого обучения, и тем, как люди думают и учатся. Дело не только в том, что люди учатся преимущественно на собственном воплощенном опыте, а не на явных обучающих примерах. Человеческий мозг устроен совершенно иначе по сравнению с дифференцируемой параметрической функцией.

Давайте мысленно отойдем подальше, окинем взором всю картину и зададимся вопросом: «В чем состоит назначение интеллекта?» Почему он возник в процессе эволюции? Мы можем лишь предполагать, но эти предположения вполне обоснованы. Допустим, можно начать с изучения мозга – органа, реализующего интеллект. Мозг – это выдающийся продукт эволюционной адаптации, уникальный механизм, постепенно развивавшийся в течение сотен миллионов лет путем случайных проб и ошибок и резко увеличивший способность организмов приспосабливаться к окружающей среде. Первоначально мозг возник более полумиллиарда лет назад как *способ хранения и выполнения поведенческих программ*. Поведенческие программы – это просто наборы инструкций, которые заставляют организм реагировать на окружающую среду: «если происходит это, значит, делай то». Они связывают сенсорные входы организма с его двигательной активностью. Вначале мозг служил средством хранения и выполнения жестко закодированных поведенческих программ (в виде паттернов нейронной связи), которые позволяли организму адекватно реагировать на поступающие сенсорные сигналы. По такому принципу до сих пор работает мозг насекомых – мух, муравьев, червей *C. elegans* (рис. 14.4) и многих других. Поскольку первоначальным «исходным кодом» этих программ служит ДНК, которая должна была быть расшифрована в паттерны нейронных связей, эволюция внезапно получила возможность осуществлять *поиск в поведенческом пространстве* практически неограниченным образом – величайший эволюционный рывок!

Эволюция была программистом, а мозг был компьютером, тщательно исполняющим код, который дала ему эволюция. Поскольку нейронная связность является очень универсальной вычислительной основой, сенсомоторное пространство всех видов, обладающих мозгом, может внезапно начать резко расширяться. Глаза, уши, нижние челюсти, 4 ноги или 24 щупальца – если у вас есть мозг, эволюция любезно разработает для вас поведенческие программы, которые будут эффективно использовать возможности тела. Мозг может справиться с любой модальностью или комбинацией модальностей, которые вы ему подбрасываете.

Но учтите, что эти древние мозги не были разумными в современном понимании. Они были биологическими автоматами и выполняли поведенческие программы, жестко закодированные в ДНК организма. Их можно назвать интеллектуальными только в том смысле, в каком является «интеллектуальным» термостат обогревателя или программа сортировки списков. Или ... обученная глубокая нейронная сеть. Это важное различие, так что давайте постараемся найти ответ на вопрос: в чем разница между автоматами и *по-настоящему интеллектуальными агентами*?



Вот в чем дело. Интеллекту присуще *понимание*, а понимание проявляется в *обобщении* – способности справиться с любой новой ситуацией, которая может возникнуть. Как отличить студента, который вызубрил ответы на экзаменационные вопросы за последние три года, но не понимает предмета, от студента, который действительно понимает материал? Предложите им решить совершенно новую задачу по теме предмета. Автомат статичен, создан для выполнения определенных действий в определенном контексте («если ... то ...»), в то время как интеллектуальный агент может на ходу адаптироваться к новым, неожиданным ситуациям. Когда автомат сталкивается с чем-то, что не соответствует «программе» (не важно, говорим ли мы о написанных человеком компьютерных программах, поведенческих программах, сгенерированных эволюцией, или о процессе неявного программирования путем подгонки модели к обучающему набору данных), он терпит неудачу. В аналогичной ситуации интеллектуальные агенты, такие как люди, максимально воспользуются навыками понимания и обобщения, чтобы найти решение.

Люди способны на большее, чем просто отображать воздействия в 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, чего мы никогда не испытывали прежде: например, нарисовать лошадь в джинсах или представить, что мы будем делать, выиграв в лотерею. Эта способность строить гипотезы, расширять пространство нашей ментальной модели за границы чувственного восприятия – 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 ключевым критерием наличия интеллекта, присущего людям и высокоразвитым животным.

Такое поведение человека резко отличается от действий автоматических систем. Жесткому автомату вообще не ведомы обобщения – он не способен справиться с чем-то, о чем ему заранее не сказали в деталях. К жестким автоматам относятся, например, хеш-таблицы или базовые программы ответов на вопросы, реализованные в виде жестко закодированных операторов *if-then-else*. Глубокие сети способны на большее: они могут успешно обрабатывать входные данные, которые немного отличаются от того, с чем они знакомы, и именно это делает их полезными. Наша модель классификации «кошка или собака» из главы 8 могла классифицировать изображения кошек или собак, которых она раньше не видела, если они были достаточно близки к тому, на чем она обучалась. Однако глубокие сети ограничиваются тем, что я называю *локальным обоб-*

щением (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* – новые вариации известных факторов, наблюдаемых во время разработки модели и широко представленных в обучающих данных, таких как разные углы камеры или условия освещения для изображений домашних животных. Это связано с тем, что глубокие сети обобщаются посредством интерполяции на многообразии (вспомните главу 5): любая вариация фактора в их входном пространстве должна быть охвачена многообразием, которое они изучают. Вот почему увеличение базовых данных так полезно для улучшения обобщающей способности глубокой сети. В отличие от людей, модели глубокого обучения не способны импровизировать в ситуациях, которые имеют лишь абстрактное сходство с прошлым опытом.

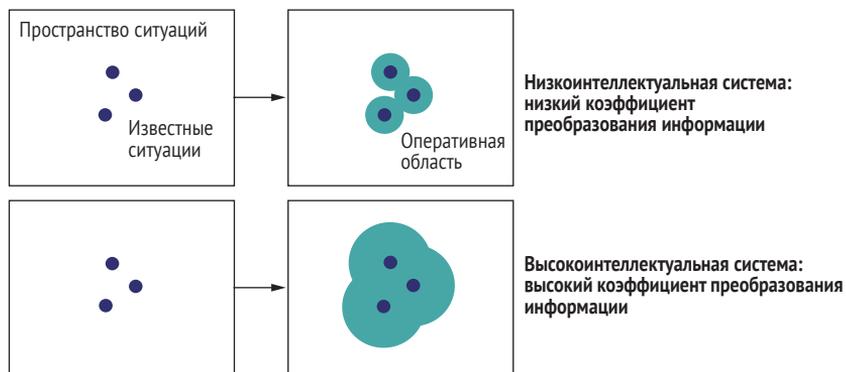


Рис. 14.5 Различие между локальным и экстремальным обобщением

Рассмотрим, например, задачу определения параметров запуска ракеты, которая должна сесть на Луну. Если для ее решения использовать глубокую сеть и обучить ее, используя 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 губительных ситуаций, пока она не научится делать вывод, что автомобили опасны, и не выработает со-

ответствующее поведение уклонения. Однако, оказавшись в другом городе, сеть вынуждена будет забыть большую часть того, что она уже изучила, и переучиваться заново. Люди, напротив, способны обучаться правилам безопасного поведения, не переживая фатального конца ни разу, – и снова благодаря своей способности абстрактного моделирования гипотетических ситуаций.

#### 14.2.4 *Предназначение интеллекта*

Различие между гибкими интеллектуальными агентами и жесткими автоматами возвращает нас к эволюции мозга. Почему мозг, который изначально был просто средством естественной эволюции для развития поведенческих автоматов, в конце концов обрел интеллект? Как и любой важный эволюционный скачок, этот качественный переход произошел лишь потому, что ему способствовал естественный отбор.

Мозг отвечает за формирование поведения. Если бы набор ситуаций, с которыми приходится сталкиваться организму, был в основном статичен и известен заранее, генерация поведения была бы простой задачей: эволюция нашла бы оптимальное поведение путем случайных проб и ошибок и жестко закодировала в ДНК организма. И действительно, эта первая стадия эволюции мозга – жесткий автомат – в свое время уже успела побывать оптимальной. Однако, что особенно важно, по мере того как сложность организма – а вместе с ней и сложность окружающей среды – продолжала возрастать, ситуации, с которыми приходилось сталкиваться животным, становились гораздо более динамичными и непредсказуемыми. Любой день вашей жизни, если вы посмотрите внимательно, не похож ни на один день, пережитый вами ранее, и не похож ни на один день, когда-либо пережитый кем-то из ваших эволюционных предков. Вы должны быть готовы постоянно сталкиваться с неизвестными и неожиданными ситуациями. Эволюция не может найти и жестко закодировать в ДНК последовательность действий, которые вы будете выполнять в течение дня, проснувшись несколько часов назад. Последовательность ваших действий генерируется на ходу каждый день.

Мозг, как инструмент формирования оптимального поведения, просто приспособился к этой потребности. Эволюция наделила мозг свойствами адаптивности и универсальности, а не просто приспособления к фиксированному набору ситуаций. Этот качественный переход, вероятно, происходил несколько раз на протяжении истории эволюции, приводя к появлению высокоинтеллектуальных животных в очень отдаленных эволюционных ветвях – человекообразных обезьян, осьминогов, воронов и других. Интеллект – это эволюционный ответ на вызовы, возникающие в сложных динамичных экосистемах.

Такова природа интеллекта: это способность эффективно использовать всю доступную информацию для формирования успешного

поведения перед лицом неопределенного, постоянно меняющегося будущего. Свойство мозга, которое Декарт называет «пониманием», является ключом к замечательной способности использовать свой прошлый опыт для разработки модульных, многообразных абстракций, которые можно быстро переназначить для обработки новых ситуаций и достижения экстремального обобщения.

## 14.2.5 Восхождение по уровням обобщения

В качестве грубой аналогии мы могли бы представить историю эволюции биологического интеллекта как медленное восхождение вверх по уровням обобщений. Все началось с мозга, похожего на автомат и способного выполнять только локальное обобщение. Со временем эволюция начала производить организмы, способные ко все более широкому обобщению, которые могли процветать во все более сложных и изменчивых средах. В конце концов, за последние несколько миллионов лет – мгновение с точки зрения эволюции – некоторые виды гоминидов достигли уровня биологического разума, способного к экстремальным обобщениям, что ускорило начало антропоцена и навсегда изменило историю жизни на Земле.

Развитие ИИ за последние 70 лет имеет поразительное сходство с эволюцией интеллекта. Ранние системы ИИ были чистыми автоматами, как чат-программа ELIZA 1960-х годов или SHRDLU<sup>1</sup> – ИИ образца 1970 года, способный манипулировать простыми объектами с помощью команд на естественном языке. В 1990-х и 2000-х годах мы стали свидетелями появления систем машинного обучения, справлявшихся с некоторым уровнем неопределенности и новизны благодаря способности к локальному обобщению. В 2010-х годах глубокое обучение еще больше расширило способность систем к локальному обобщению, позволив инженерам использовать гораздо большие наборы данных и гораздо более выразительные модели.

Вполне возможно, что сегодня мы стоим на пороге следующего эволюционного перехода. Растет интерес к системам, которые могут достичь *широкого обобщения*, которое я определяю как способность иметь дело с *неизвестными неизвестными* в рамках одной широкой области задач (включая ситуации, с которыми система не была обучена справляться и которые ее создатели не могли предвидеть). Это может быть, например, беспилотный автомобиль, способный безопасно справляться с любой ситуацией, которую вы ему подбираете, или домашний робот, который может пройти «тест Возняка на интеллект» – зайти на случайную кухню и приготовить чашку кофе<sup>2</sup>. Сочетая глубокое обучение и кропотливо созданные вручную

<sup>1</sup> Terry Winograd, *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language* (1971).

<sup>2</sup> Fast Company, *Wozniak: Could a Computer Make a Cup of Coffee?* (March 2010), <http://mng.bz/pJMP>.

абстрактные модели мира, исследователи уже добились заметного прогресса в достижении этих целей.

Однако на данный момент ИИ ограничивается лишь когнитивной автоматизацией: использование термина «интеллект» здесь является существенным преувеличением. Было бы правильнее назвать эту область исследований «искусственным познанием», где «когнитивная автоматизация» и «искусственный интеллект» были бы двумя почти независимыми отраслями. С этой точки зрения «искусственный интеллект» – огромная и почти нетронутая область исследований, где многое предстоит открыть или изобрести. Я не хочу приуменьшать достижения глубокого обучения. Когнитивная автоматизация невероятно полезна, и способность моделей глубокого обучения автоматизировать задачи, основываясь только на данных, представляет собой особенно мощную форму когнитивной автоматизации, гораздо более практичную и универсальную, чем явное программирование. Качественно сделанная когнитивная автоматизация меняет правила игры практически для любой отрасли. Но до человеческого (или животного) интеллекта нам еще далеко. Пока что наши модели могут выполнять только локальное обобщение: они отображают пространство  $X$  в пространство  $Y$  с помощью плавного геометрического преобразования, извлеченного из плотной выборки точек данных  $X$  в  $Y$ , и любое нарушение в пространстве  $X$  или  $Y$  делает это отображение недействительным. Они могут обобщать только ситуации, которые похожи на прошлые данные, тогда как человеческое познание способно к экстремальным обобщениям, быстрой адаптации к радикально новым ситуациям и долгосрочному планированию.

## 14.3 Курс на большую универсальность в ИИ

Чтобы устранить ограничения, о которых мы говорили выше, и создать ИИ, способный конкурировать с человеческим мозгом, нам нужно отойти от простого сопоставления ввода-вывода и перейти к рассуждениям и абстракциям. В следующих нескольких разделах мы обсудим, как может выглядеть путь к решению этой непростой задачи.

### 14.3.1 О важности постановки правильной цели: правило короткого пути

Биологический интеллект был ответом на вопрос, заданный природой. Аналогично, если мы хотим разработать настоящий искусственный интеллект, нам нужно научиться задавать правильные вопросы.

Эффект, который мы постоянно наблюдаем при проектировании систем глубокого обучения, – это *правило короткого пути* (shortcut

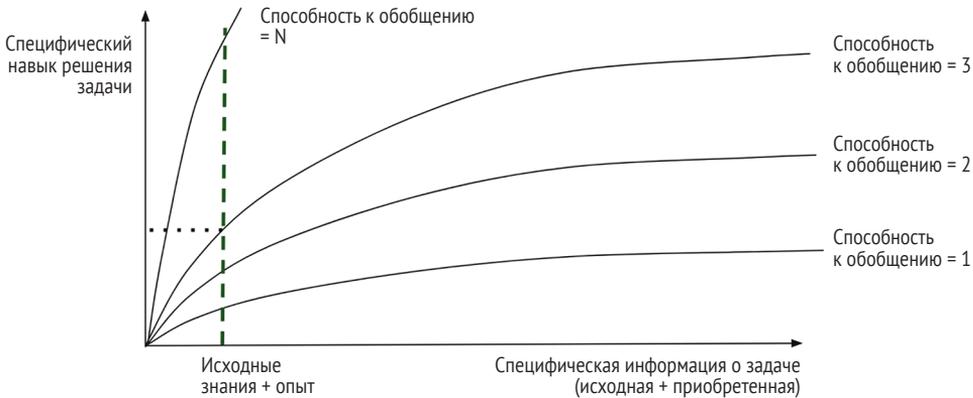
rule). Если вы сосредоточитесь на оптимизации одной очевидной метрики успеха (пойдете коротким путем), то достигнете своей цели, но в ущерб тем частям системы, которые не были учтены вашей метрикой. В конечном итоге вы используете все доступные короткие пути к цели. Окончательный вид вашего творения определяют цели, которые вы перед собой ставите.

Неудачная постановка целей очень часто встречается на соревнованиях по машинному обучению. В 2009 году компания Netflix провела конкурс, в котором команде, набравшей наибольшее количество баллов за систему рекомендации фильмов, был обещан приз в 1 млн долларов. В итоге систему, созданную победившей командой, так и не удалось внедрить в производство, потому что она была слишком сложной и требовательной к вычислительным ресурсам. Победители пошлись по короткому пути и сосредоточились только на точности предсказания – цель, к которой они стремились, – в ущерб всем остальным важным характеристикам системы: стоимости логического вывода, простоты обслуживания, надежности и объяснимости. Правило короткого пути срывает и в большинстве соревнований Kaggle: модели, созданные победителями состязаний, редко, если вообще когда-либо, внедряются в производство.

Правило короткого пути повсеместно применяется в области ИИ на протяжении нескольких последних десятилетий. В 1970-х годах психолог и пионер информатики Аллен Ньюэлл, обеспокоенный тем, что его область науки не достигает сколько-нибудь значимого прогресса в направлении правильной теории познания, предложил новую великую цель для ИИ: научиться играть в шахматы. Ньюэлл исходил из того, что игра в шахматы очевидно требует наличия у людей таких способностей, как восприятие, рассуждение и анализ, память, изучение книг и т. д. Следовательно, если мы сумеем построить машину для игры в шахматы, она также будет обладать этими способностями, не так ли? Более двух десятилетий спустя мечта Ньюэлла сбылась – в 1997 году суперкомпьютер Deep Blue от IBM обыграл Гарри Каспарова, лучшего шахматиста в мире. Но вот беда – исследователи обнаружили, что создание искусственного чемпиона по шахматам мало что рассказало им о человеческом интеллекте. Алгоритм Alpha-Beta, лежащий в основе Deep Blue, не был моделью человеческого мозга и не подходил для других задач, кроме похожих на шахматы настольных игр. Оказалось, что создать ИИ, способный эффективно играть в шахматы, было проще, чем создать искусственный разум, поэтому создатели Deep Blue невольно соскользнули на короткий путь.

До сих пор основным показателем успеха в области ИИ было решение конкретных задач, от шахмат до игры в го, от классификации MNIST до ImageNet, от аркадных игр Atari до StarCraft и Dota 2. Следовательно, вся история «успехов» в области искусственного разума сводится к тому, что разработчики придумали, как решать конкретные задачи, не используя подлинный интеллект.

Если это утверждение вас шокирует, имейте в виду, что ключевым свойством человеческого интеллекта не является наличие навыков решения какой-либо конкретной задачи – напротив, это способность адаптироваться к новизне, эффективно приобретать новые навыки и решать невиданные ранее задачи. В случае фиксированной задачи мы вырабатываем точный способ решения либо путем жесткого кодирования предоставленных человеком знаний, либо путем изучения огромных объемов данных. Вы даете инженерам возможность «купить» больше навыков для своего ИИ, просто добавляя данные или жестко запрограммированные знания, не увеличивая при этом обобщающую способность ИИ (рис. 14.6). Если у вас есть почти бесконечные обучающие данные, даже очень грубый алгоритм, такой как поиск ближайшего соседа, сможет научиться играть в видеоигры со сверхчеловеческими способностями. Аналогичного результата можно добиться при помощи огромного количества написанных человеком операторов `if-then-else`. Но этот подход работает ровно до тех пор, пока вы не внесете небольшое изменение в правила игры – такое, к которому человек может мгновенно приспособиться, – что потребует переобучения или перестройки неинтеллектуальной системы с нуля.



**Рис. 14.6 Система с низким уровнем обобщения может достичь произвольно сложного навыка решения фиксированной задачи при наличии неограниченного количества специфической для задачи информации**

Короче говоря, фиксируя условия задачи, вы устраняете необходимость справляться с неопределенностью и новизной, а поскольку природа интеллекта заключается именно в способности справляться с неопределенностью и новизной, вы эффективно устраняете потребность в интеллекте. И поскольку всегда проще найти автоматизированное решение конкретной задачи, чем решить общую проблему интеллекта, это кратчайший путь, который вы выберете в 100 % случаев. Люди могут использовать свой обобщенный интеллект для приобретения специальных навыков, но в обратную сто-

рону это не работает – нет пути от набора специальных навыков к общему интеллекту.

### 14.3.2 Новая цель

Чтобы сделать искусственный интеллект действительно разумным и научить его справляться с невероятной изменчивостью реального мира, нам сначала нужно отказаться от стремления к достижению *навыков, специфичных для конкретной задачи*, и вместо этого сосредоточиться на способности к обобщению как таковой. Нам нужны новые индикаторы прогресса, которые помогут нам разрабатывать все более интеллектуальные системы, новые метрики, которые укажут правильное направление и дадут эффективный сигнал обратной связи. Пока мы ставим перед собой цель «создать модель, решающую задачу X», будет срабатывать правило кратчайшего пути, и в итоге мы получим модель, решающую задачу X, – и только.

На мой взгляд, интеллект может быть точно определен количественно как *коэффициент эффективности*: своего рода коэффициент соответствия между *объемом доступной вам значимой информации* о мире (которая может быть либо прошлым опытом, либо врожденными предварительными знаниями) и *широтой новой области деятельности* – набором новых ситуаций, в которых вы сможете вести себя правильно (вы можете рассматривать это как свой *набор навыков*). Более интеллектуальный агент сможет справиться с более широким набором будущих задач и ситуаций, используя меньший объем прошлого опыта. Чтобы измерить такое соответствие, вам просто нужно зафиксировать информацию, доступную вашей системе, – ее опыт и ее предшествующие знания – и оценить эффективность системы на наборе эталонных ситуаций или задач, существенно отличающихся от тех, с которыми она имела дело раньше. Стремление максимизировать это соотношение должно привести нас к созданию подлинного искусственного интеллекта. Чтобы избежать самобмана, крайне важно тестировать систему только на задачах, для которых она не была запрограммирована или обучена, – в идеале нам нужны задачи, которые создатели системы не могли даже предвидеть.

В 2018 и 2019 годах я разработал эталонный набор данных под названием «Корпус абстракций и логических умозаключений» (Abstraction and Reasoning Corpus, ARC<sup>1</sup>), который призван отразить это определение интеллекта. Тестовый набор ARC задуман как доступный в равной степени машинам и людям, и он очень похож на тесты человеческого IQ, такие как *прогрессивные матрицы Равена*. Во время тестирования перед вами ставят ряд «задач». Каждая задача объ-

---

<sup>1</sup> François Chollet, *On the Measure of Intelligence* (2019), <https://arxiv.org/abs/1911.01547>.

ясняется с помощью трех или четырех «примеров», оформленных в виде набора входов и соответствующих им выходов (рис. 14.7). Затем вам предъявляют совершенно новый набор входов, и у вас будет три попытки создать правильный набор выходов, прежде чем перейти к следующему заданию.

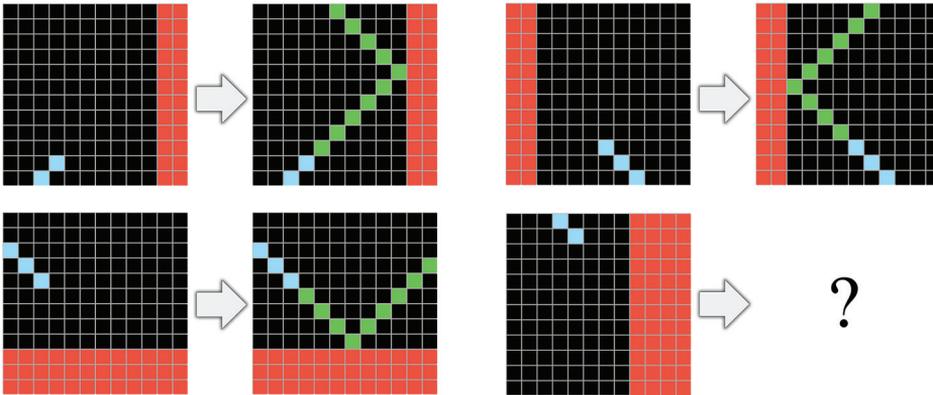


Рис. 14.7 Задача ARC. Характер задачи демонстрируется парой примеров пар ввода-вывода. Имея новый ввод, вы должны построить соответствующий вывод

Набор заданий ARC отличается от традиционных тестов IQ двумя факторами. Во-первых, ARC стремится измерить способность к обобщению, проверяя вас только на задачах, с которыми вы никогда раньше не сталкивались. Это означает, что ARC – это игра, которой вы не можете научиться, по крайней мере теоретически: каждое тестовое задание имеет свою уникальную логику, которую вам придется понимать на лету. Вы не можете просто запомнить конкретные стратегии из прошлых задач.

Во-вторых, ARC старается ограничить использование накопленных знаний, которые вы приносите на тест. На практике вы никогда не подходите к новой проблеме полностью с нуля – вы используете приобретенные навыки и информацию. ARC исходит из того, что все испытуемые должны начинать с набора накопленных знаний, называемых «априорными базовыми знаниями», т. е. «системы знаний», с которой люди рождаются. В отличие от теста на IQ, задачи ARC стараются исключить приобретенные знания, такие как, например, грамматика предложений человеческого языка.

Неудивительно, что методы, основанные на глубоком обучении (включая модели, обученные на очень больших объемах внешних данных, например GPT-3), оказались совершенно неспособными решать задачи ARC, поскольку они не представляют интерполяции в пространстве гипотез и плохо поддаются аппроксимации. Между тем обычным людям не составляет труда решить эти задачи с первого раза, без какой-либо тренировки. Когда мы видим, как дети в возрасте пяти лет естественным образом решают задачи, совершенно

непосильные для современных технологий искусственного интеллекта, это четкий сигнал о том, что мы упускаем нечто важное.

Что потребуется для решения задач ARC? Надеюсь, этот вопрос заставит вас задуматься. В этом весь смысл ARC: дать вам цель другого рода, заставить уйти с кратчайшего пути и двинуться в новом направлении – надеюсь, оно будет достаточно продуктивным. Теперь давайте кратко рассмотрим ключевые ингредиенты, которые вам понадобятся, если вы хотите ответить на этот вызов.

## 14.4 Реализация интеллекта: недостающие ингредиенты

Итак, мы пришли к выводу, что интеллект – это гораздо больше, чем скрытая интерполяция по многообразию, которую выполняет глубокое обучение. Но что же тогда нужно, чтобы начать создавать настоящий интеллект? Каковы ключевые аспекты, которые в настоящее время ускользают от нас?

### 14.4.1 Построение и использование абстрактных аналогий

Интеллект – это способность использовать свой прошлый опыт (и врожденные базовые знания), чтобы противостоять новым, неожиданным будущим ситуациям. Если будущее, с которым вам предстоит столкнуться, будет *действительно новым*, не имеющим ничего общего с тем, что вы видели раньше, вы не сможете отреагировать на него, каким бы умным вы ни были.

Интеллект работает, потому что для нас не существует абсолютно беспрецедентных явлений. Столкнувшись с чем-то новым, мы применяем аналогию с прошлым опытом, формулируя его в терминах абстрактных понятий, которые мы накопили за прошедшее время. Человек из XVII века, впервые увидевший реактивный самолет, мог бы описать его как большую и шумную металлическую птицу, которая не машет крыльями. Автомобиль? Ну конечно, это безлошадная повозка. Преподавая детям физику, вы можете объяснить, что электричество похоже на воду в трубе или что пространство-время похоже на резиновый лист, который искажается тяжелыми предметами.

Помимо таких четких, явных аналогий, мы постоянно проводим более мелкие, неявные аналогии, каждую секунду, с каждой мыслью. Аналогии – это наш способ ориентироваться в жизни. Оказались в новом супермаркете? Вы успешно совершите покупки, построив аналогию с другими магазинами, в которых вы были раньше. Довелось беседовать с новым коллегой? Ваш разговор будет построен на аналогиях с другими людьми. Даже кажущиеся случайными узоры, вроде формы облаков, мгновенно вызывают в нас яркие образы – слона, корабля, рыбы.

Эти аналогии существуют не только в нашем сознании: физическая природа полна изоморфизмов. Электромагнетизм аналогичен гравитации. Все животные структурно похожи друг на друга из-за общего происхождения. Кристаллы кремнезема похожи на кристаллы льда. И т. д.

Я называю это явление *гипотезой калейдоскопа*: кажется, что наше восприятие мира отличается невероятной сложностью и нескончаемой новизной, но все в этом море сложности похоже на все остальное. Количество *уникальных атомов смысла*, которые вам нужны для описания окружающего мира, относительно невелико, и все вокруг вас представляет собой рекомбинацию этих атомов, бесконечные вариации нескольких компонентов – очень похоже на то, что происходит внутри калейдоскопа, где несколько стеклянных кристаллов отражаются системой зеркал, создавая богатые, казалось бы, бесконечно меняющиеся узоры (рис. 14.8).

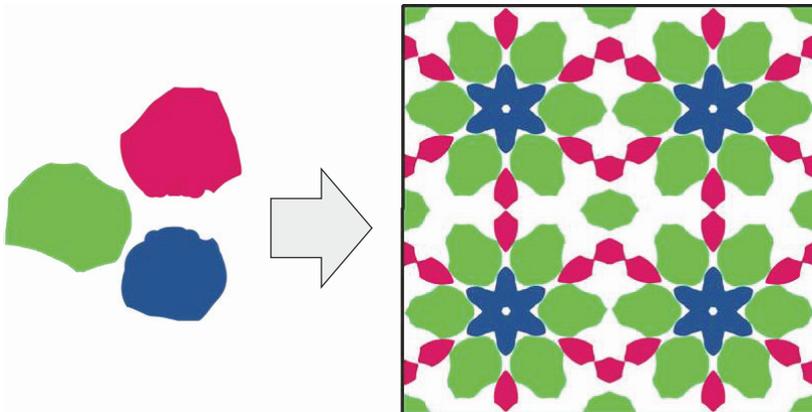


Рис. 14.8 Калейдоскоп создает богатые (но повторяющиеся) узоры всего из нескольких кусочков цветного стекла

Способность к обобщению, или интеллект, – это способность анализировать свой опыт, чтобы извлечь атомы смысла, которые, по-видимому, можно повторно использовать во многих различных ситуациях. После извлечения они становятся *абстракциями*. Всякий раз, столкнувшись с новой ситуацией, вы осмысливаете ее с помощью накопленной коллекции абстракций. Но как наш мозг извлекает атомы смысла, пригодные к повторному использованию? Просто замечая, когда две вещи похожи, замечая аналогии. Если что-то повторяется, то оба явления должны иметь одно начало, как в калейдоскопе. Абстракция – это топливо интеллекта, а построение аналогий – это источник абстракции.

Короче говоря, *интеллект – это восприимчивость к абстрактным аналогиям*, и это, по сути, исчерпывающее определение. Наличие высокой восприимчивости к аналогиям позволяет нам извлекать

мощные абстракции из ограниченного опыта и использовать их в обширной области пространства будущего опыта. Иными словами, мы умеем максимально эффективно преобразовывать прошлый опыт в способность справляться с новизной будущего.

## 14.4.2 Два полюса абстракции

Если интеллект – это восприимчивость к аналогиям, то разработка искусственного интеллекта должна начинаться с разработки пошагового алгоритма проведения аналогий. Аналогия начинается со *сравнения сущностей друг с другом* (под сущностями здесь мы понимаем все наблюдаемые и осмысливаемые нами объекты и явления). Важно отметить, что есть *два разных способа* сравнивать сущности, из которых возникают два разных вида абстракции, два способа мышления, каждый из которых лучше подходит для решения определенных задач. Вместе эти два полюса абстракции составляют основу нашего интеллекта.

Первый способ связать сущности друг с другом – это *сравнение на уровне сходства*, которое порождает *анalogии свойств*. Вторым способом – *точное структурное соответствие*, порождающее *программные аналогии* (или *структурные аналогии*). В обоих случаях вы начинаете с экземпляров сущности и объединяете связанные аналогией экземпляры в своего рода кластер, чтобы создать абстракцию. Различие между полюсами заключается в том, как вы сообщаете, что два экземпляра связаны, и как вы объединяете экземпляры в абстракции. Рассмотрим подробнее каждый тип аналогии.

### Аналогия, основанная на свойствах

Допустим, вы нашли на своем заднем дворе несколько жуков, принадлежащих к разным видам. Вы заметите сходство между ними. Некоторые будут более похожи друг на друга, а некоторые будут менее похожи: понятие сходства неявно представляет собой гладкую непрерывную *функцию расстояния*, определяющую скрытое многообразие, в котором обитают ваши экземпляры. Насмотревшись на достаточное количество жуков, вы можете начать группировать похожие экземпляры и объединять их в набор *прототипов*, который отражает общие визуальные особенности каждого кластера (рис. 14.9). Этот прототип является абстрактным: он не совпадает ни с одним конкретным экземпляром, который вы видели, зато он кодирует свойства, общие для всех них. Встретив нового жука, вы не станете сравнивать его с каждым экземпляром жука, встреченного раньше, чтобы решить, что с ним делать. Вам достаточно сравнить нового жука с несколькими известными прототипами, чтобы найти наиболее похожий – *категорию жука* – и использовать его для полезных прогнозов: может ли он вас укусить? Причинит ли он вред вашей яблоне?



Рис. 14.9. Аналогия на уровне свойств связывает экземпляры через непрерывное подобие и позволяет создать абстрактные прототипы

Звучит знакомо, не так ли? Действительно, это очень похоже на то, как работает машинное обучение без учителя (например, алгоритм кластеризации  $k$ -средних). В целом все современное машинное обучение, с учителем или нет, работает путем изучения скрытых многообразий, описывающих пространство экземпляров, закодированных с помощью прототипов. (Помните признаки сверточной сети, которые вы визуализировали в главе 9? Это были визуальные прототипы.) Аналогия, ориентированная на свойство (или признак в терминах машинного обучения), – это вид аналогии, который позволяет моделям глубокого обучения выполнять локальное обобщение.

На этой аналогии также основаны многие ваши собственные когнитивные способности. Как человек вы все время проводите ценностно-ориентированные аналогии. Это разновидность абстракции, лежащая в основе распознавания образов, восприятия и интуиции. Если вы можете решить задачу, не задумываясь о ней, вы в значительной степени опираетесь на аналогии, связанные со свойствами. Например, если вы смотрите фильм и начинаете подсознательно классифицировать разных персонажей по «типам», это абстракция, основанная на свойствах объекта.

## АНАЛОГИЯ, ОСНОВАННАЯ НА ПРОГРАММЕ

Важно отметить, что познание – это нечто большее, чем непосредственная, приблизительная, интуитивная категоризация, основанная на аналогии свойств. Есть еще одна разновидность механизма построения абстракций, более медленного, точного и обдуманного: программная (или структурная) аналогия.

Если вам доводилось заниматься разработкой компьютерных программ, наверняка вы создавали разные функции или классы, которые

имеют между собой много общего. Обнаружив такую избыточность, вы задавали себе вопрос «Можно ли создать более абстрактную функцию, выполняющую ту же работу и пригодную для многократного использования? Может ли существовать абстрактный базовый класс, от которого наследуют оба моих класса?» Определение абстракции, которое вы здесь используете, основано на программной аналогии. Вы не пытаетесь сравнить *похожесть* своих классов и функций с помощью неявной функции расстояния, как вы поступаете при сравнении человеческих лиц. Скорее, вас интересует, есть ли в них части, имеющие *одинаковую функциональную структуру*. Вы ищете то, что называется *изоморфизмом подграфов* (рис. 14.10). Программы могут быть представлены в виде графов операторов, и вы пытаетесь найти совпадающие подграфы (подмножества программ).

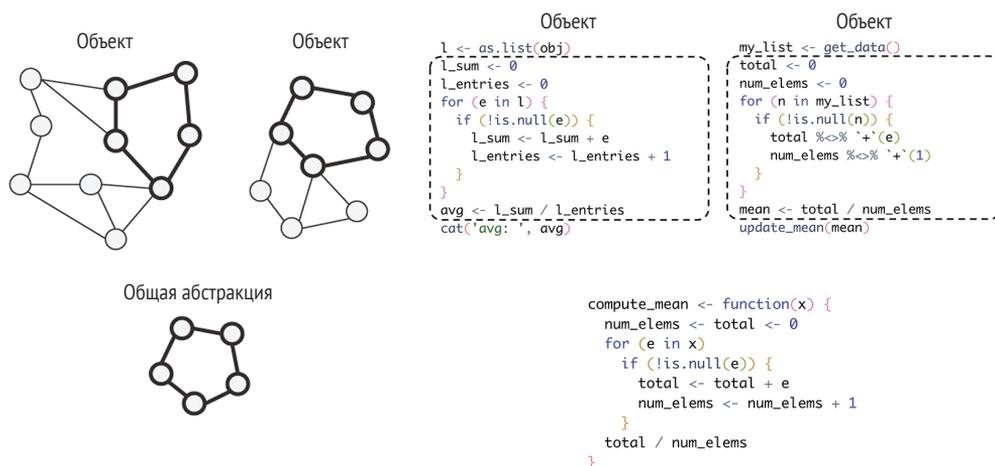


Рис. 14.10 Программная аналогия обнаруживает и выделяет изоморфные подструктуры в разных экземплярах

Этот способ построения аналогий посредством поиска точного структурного соответствия в различных дискретных структурах вовсе не ограничивается такими областями, как информатика или математика – вы постоянно используете его, сами того не замечая. Он лежит в основе рассуждений, планирования и общей концепции *строгого вывода* (в отличие от интуиции). Всякий раз, когда вы думаете об объектах, связанных друг с другом дискретной сетью отношений (а не непрерывной функцией подобия), вы используете программные аналогии.

### ПЗНАНИЕ КАК ОБЪЕДИНЕНИЕ ПОЛЮСОВ АБСТРАКЦИИ

Для наглядности сопоставим два полюса абстракции (табл. 14.1).

Таблица 14.1. Два полюса абстракции

Абстракция на основе свойств	Программная абстракция
Связывает вещи через функцию расстояния	Связывает вещи через совпадение структур
Непрерывная, основана на геометрии	Дискретная, основана на топологии
Образует абстракции как «среднее» всех сущностей, принадлежащих к одному прототипу	Образует абстракции путем выделения изоморфных подструктур
Лежит в основе восприятия и интуиции	Лежит в основе рассуждений и планирования
Непосредственная, гибкая, приближительная	Непрямая, точная, строгая
Требует наличия значительного опыта для достижения полезных результатов	Эффективно использует небольшой опыт, может опираться буквально на пару случаев

### 14.4.3 Сочетание двух полюсов абстракции

Все, что мы делаем, все, о чем мы думаем, представляет собой сочетание двух типов абстракции. Вряд ли вы найдете задачу, в которой будет задействован только один тип абстракции. Даже такая, казалось бы, задача «чистого восприятия», как распознавание объектов на визуальной сцене, требует изрядных рассуждений об отношениях между объектами, на которые вы смотрите. Даже решение такой, казалось бы, «чисто аналитической» задачи, как поиск доказательства математической теоремы, не обходится без значительной доли интуиции. Еще до того, как математик коснется пером бумаги, у него уже есть смутное представление о направлении, в котором он движется. Дискретные логические шаги, которые он предпринимает, чтобы добраться до цели, обусловлены интуицией высокого уровня.

Эти два полюса дополняют друг друга, и именно их чередование делает возможным чрезвычайно мощное обобщение. Ни один разум не сможет обойтись только одной разновидностью абстракции.

### 14.4.4 Недостающая половина картинки

К этому моменту вы наверняка начали понимать, чего не хватает современному глубокому обучению: оно очень хорошо кодирует абстракцию, основанную на свойствах, но практически не способно генерировать абстракцию, основанную на программах и структуре. Человеческий интеллект представляет собой сложное чередование обоих типов абстракции, значит, при разработке ИИ мы до сих пор упускали половину того, что нам нужно, – возможно, более важную половину. Но хочу предостеречь вас от заблуждения. До сих пор я представлял оба типа абстракций как полностью обособленные друг от друга – даже противоположные. На практике, однако, это скорее *спектр абстракций*: в какой-то степени вы можете рас-

суждать, встраивая дискретные программы в непрерывные многообразия, точно так же, как вы можете подогнать полиномиальную функцию к почти любому набору дискретных точек, если у вас достаточно много коэффициентов. И наоборот, вы можете использовать дискретные программы для эмуляции функций непрерывного расстояния – в конце концов, когда вы выполняете вычисления линейной алгебры на компьютере, вы работаете с непрерывными пространствами, используя исключительно дискретные программы, оперирующие единицами и нулями.

Разумеется, существуют задачи, которые лучше подходят для одного или другого типа абстракции. Например, попробуйте научить модель глубокого обучения сортировать список всего из пяти произвольных чисел. С правильной архитектурой это возможно, но фактически выльется в сплошное разочарование. Чтобы достичь успеха, вам понадобится огромный объем обучающих данных, и даже в этом случае модель все равно будет время от времени совершать ошибки при сортировке новых чисел. А если вы захотите освоить сортировку списков из 10 чисел, вам потребуется полностью переобучить модель на еще большем количестве данных. Между тем программа сортировки чисел на  $R$  занимает всего несколько строк и после проверки на паре примеров будет безотказно работать со списками любого размера. Переход от пары демонстрационных и тестовых примеров к программе, которая успешно обрабатывает любой список чисел, является чрезвычайно сильным обобщением.

И наоборот, задачи восприятия чрезвычайно плохо решаются при помощи дискретных рассуждений. Попробуйте написать программу на чистом  $R$  для классификации цифр MNIST без использования каких-либо методов машинного обучения – вас ждет путешествие в ад. Вы будете сутки напролет кропотливо создавать функции, способные определить количество замкнутых циклов в цифре, координаты центра масс цифры и т. д. Очнувшись после написания тысяч строк изнуряющего кода, вы обнаружите, что еле-еле достигли точности 90 % на проверочном наборе. В этом случае намного проще обучить параметрическую модель, которая лучше использует большой объем доступных данных и достигает гораздо более надежных результатов. Если у вас есть много данных и вы столкнулись с ситуацией, когда применима гипотеза многообразия, используйте глубокое обучение.

По этой причине маловероятно, что в будущем получит развитие подход, который сводит задачу логического вывода исключительно к интерполяции на многообразии или ограничивает проблемы восприятия только дискретными рассуждениями. Перспектива развития искусственного интеллекта заключается в разработке единой структуры, которая включает в себя *оба* типа абстрактных аналогий. Давайте попытаемся представить, как она может выглядеть.

## 14.5 Будущее глубокого обучения

Зная, как действуют глубокие сети, и понимая текущее положение дел в сфере исследований, можем ли мы предсказать направление движения в среднесрочной перспективе? Далее приводятся некоторые мои личные мысли. Имейте в виду, что у меня нет хрустального шара, поэтому многим моим ожиданиям, может быть, не суждено стать реальностью. Я разделяю эти прогнозы не потому, что их состоятельность будет доказана в ближайшем будущем, а потому, что они интересны и выглядят реальными в настоящем.

Вот основные направления, которые мне кажутся многообещающими:

- *модели, более близкие к универсальным компьютерным программам*, построенные на основе более широкого набора примитивов, чем современные дифференцируемые слои. Именно так мы приблизимся к возможности моделирования рассуждений и обобщения, отсутствие которой является основным недостатком современных моделей;
- *сочетание 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, что эти соображения не относятся к какому-то конкретному виду обучения с учителем, которое до сих пор остается хлебом насущным глубокого обучения, скорее, они применимы к любой форме машинного обучения, включая обучение без учителя и самообучение, а также обучение с подкреплением. Принципиально не важно, откуда берутся размеченные данные или как выглядит цикл обучения; это всего лишь разные ветви машинного обучения – разные грани одной и той же конструкции. Давайте рассмотрим их поближе.

### 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.

Это положение дел может очень сильно измениться благодаря *синтезу программ* – области, которая сегодня занимает лишь узкую нишу, но может буквально «взлететь» в ближайшие десятилетия. 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. Отличие в том, что вместо обучения значений параметров в четко определенной программе (нейронной сети) мы генерируем исходный код посредством процесса дискретного поиска (табл. 14.2).

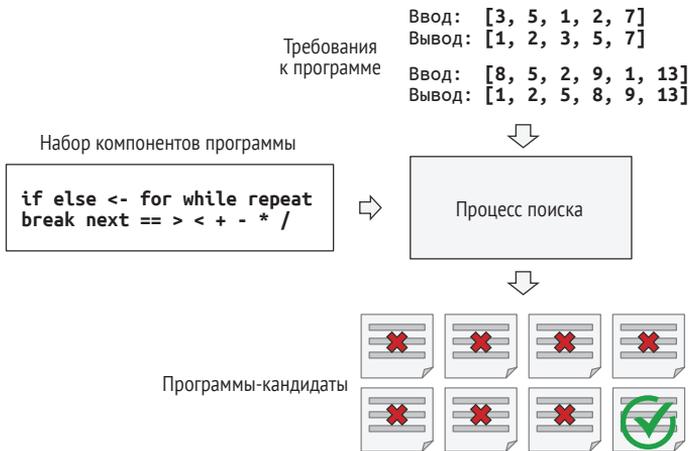


Рис. 14.11 Схематическое представление синтеза программы. Исходя из заданной спецификации программы и набора компонентов, процесс поиска собирает из стандартных компонентов программы-кандидаты, которые затем проверяются на соответствие спецификации. Поиск продолжается до тех пор, пока не будет найдена подходящая программа

Таблица 14.2. Машинное обучение и синтез программ

Машинное обучение	Синтез программ
Модель: дифференцируемые параметрические функции	Модель: граф операторов на языке программирования
Механизм: градиентный спуск	Механизм: дискретный поиск (такой как генетический поиск)
Нуждается в большом объеме обучающих данных для достижения приемлемых результатов	Не требователен к данным; может работать с незначительным количеством обучающих примеров

### 14.5.2 Машинное обучение и синтез программ

Синтез программ – это способ добавить возможности программной абстракции в наши системы искусственного интеллекта. Это недостающая часть головоломки. Ранее я упоминал, что методы глубокого обучения совершенно непригодны для решения тестов ARC, ориентированных на логический вывод и абстракцию. Между тем даже первые эксперименты в области синтеза программ уже демонстрируют весьма многообещающие результаты при решении этого теста.

### 14.5.3 Сочетание глубокого обучения и синтеза программ

Разумеется, синтез программ не является заменой глубокого обучения – это его дополнение. Это второе полушарие, которого до сих пор не доставало нашему искусственному мозгу. Нам предстоит научиться использовать комбинацию разных подходов по двум направлениям:

- 1 разработка систем, объединяющих как модули глубокого обучения, так и дискретные алгоритмические модули;
- 2 использование глубокого обучения для повышения эффективности самого процесса поиска программы.

Рассмотрим эти направления более подробно.

#### **ИНТЕГРАЦИЯ МОДУЛЕЙ ГЛУБОКОГО ОБУЧЕНИЯ И АЛГОРИТМИЧЕСКИХ МОДУЛЕЙ В ГИБРИДНЫЕ СИСТЕМЫ**

Сегодня самые мощные системы искусственного интеллекта являются гибридными: они используют как модели глубокого обучения, так и созданные вручную программы манипулирования символами. Например, в игре AlphaGo компании DeepMind большая часть интеллекта спроектирована и запрограммирована опытными программистами с применением четких алгоритмов (таких как алгоритм поиска Монте-Карло в деревьях); обучение на данных происходит только в специализированных модулях (оценочные и стратегические сети). Или возьмем автономные транспортные средства: беспилотный автомобиль способен справляться с самыми разными ситуациями, потому что он формирует модель окружающего мира – буквально трехмерную модель, – полную предположений, жестко закодированных инженерами-людьми. Эта модель постоянно обновляется с помощью модулей восприятия глубокого обучения, которые связывают ее с окружением автомобиля.

Для обеих этих систем – AlphaGo и автономных транспортных средств – комбинация созданных человеком дискретных программ и обученных непрерывных моделей открывает качественно новый уровень, который был бы невозможен при использовании любого из этих подходов по отдельности. Пока что отдельные алгоритми-

ческие элементы таких гибридных систем тщательно разработаны инженерами-людьми. Но в будущем подобные системы можно будет полностью обучить без участия человека.

Как это будет происходить? Рассмотрим хорошо изученный тип сетей: рекуррентные нейронные сети (RNN). Важно отметить, что RNN имеют немного меньше ограничений, чем сети прямого пространства, 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 обширный набор программных примитивов, которыми модель 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 определенные модули останутся дифференцируемыми, и 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, и геометрических модулей, поддерживающих неформальное знание и распознавание образов (рис. 14.12). Вся система будет обучаться практически без участия человека. Это должно резко расширить круг задач, которые можно решить с помощью машинного обучения, – пространство программ, которые мы можем генерировать автоматически на основе обучающих данных. Такие системы, как AlphaGo или даже RNN, можно считать доисторическими предками гибридных алгоритмически-геометрических моделей.



Рис. 14.12 Обученная программа, основанная одновременно как на геометрических примитивах (распознавание шаблонов, предсказание), так и на алгоритмических примитивах (рассуждение, поиск, память)

## ИСПОЛЬЗОВАНИЕ ГЛУБОКОГО ОБУЧЕНИЯ ДЛЯ НАПРАВЛЕННОГО ПОИСКА ПРОГРАММ

Сегодня синтез программ сталкивается с серьезным препятствием: он чрезвычайно неэффективен. Если говорить очень упрощенно, синтез программ перебирает каждую возможную программу в пространстве поиска, пока не найдет ту, которая соответствует предоставленной спецификации. По мере увеличения сложности спецификации программы или расширения словаря примитивов, используемых для написания программ, процесс поиска программы сталкивается с так называемым *комбинаторным взрывом*, когда набор программ-кандидатов растет чрезвычайно быстро – даже намного быстрее, чем просто экспоненциально. Поэтому сегодня синтез программ подходит для генерации только очень простого и короткого кода. В обозримом будущем вы не сможете создать таким прямолинейным способом новую операционную систему для своего компьютера.

Чтобы достичь заметного прогресса, нам нужно повысить эффективность синтеза программ, сделав его похожим на человеческий подход к программированию. Когда вы открываете редактор, чтобы написать код, вы не думаете о каждой возможной программе, которую потенциально могли бы разработать. Вы ограничиваете себя очень узкими рамками, используя свое понимание проблемы и прошлый опыт, чтобы резко сократить пространство возможных вариантов для рассмотрения.

Глубокое обучение способно сделать то же самое с синтезом программ: хотя каждая конкретная программа, которую мы могли бы сгенерировать, может быть принципиально дискретным объектом, не выполняющим интерполяцию данных, пространство всех полезных программ может быть очень похоже на непрерывное многообразие. Это означает, что модель глубокого обучения, обученная на миллионах успешных эпизодов генерации программ, может начать генерировать надежные подсказки о направлении поиска в пространстве программ на пути от спецификации к ожидаемой программе – точно так же, как программист сначала оценивает общую

архитектуру программы, которую он собирается написать, и обдумывает, какие промежуточные функции и классы он будет использовать в качестве ступенек на пути к цели.

Напомню, что человеческое мышление преимущественно руководствуется абстракцией на основе свойств, то есть распознаванием образов и интуицией. Синтез программ должен вести себя так же. Я предполагаю, что общий подход к управлению поиском программ с помощью изученной эвристики станет предметом растущего интереса исследователей в течение следующих 10–20 лет.

#### 14.5.4 *Непрерывное обучение и повторное использование модульных подпрограмм*

Когда модели станут сложнее и будут основаны на более насыщенных алгоритмических примитивах, эта повышенная сложность потребует увеличить степень повторного использования результатов прежних решений вместо обучения новых моделей с нуля каждый раз, когда возникает новая задача или новый набор данных. Многие наборы данных содержат недостаточно информации, чтобы мы могли приступить к созданию новых, сложных моделей с нуля, и поэтому необходимо будет использовать информацию из прежних наборов данных (представьте, что вам пришлось бы изучать русский язык с нуля всякий раз, когда вы открываете новую книгу, – это было бы просто невозможно). Обучение моделей с нуля для каждой новой задачи неэффективно также из-за большого перекрытия между текущими задачами и прежними.

В последние годы неоднократно отмечалось интересное явление: обучение *одной и той же модели* для решения мало связанных между собой задач дает в результате модель, которая *лучше подходит для каждой задачи*. Например, обучение одной и той же нейронной модели машинного перевода с английского на немецкий и с французского на итальянский дает в результате модель, которая лучше подходит для каждой пары языков. Аналогично, одновременное обучение модели классификации и сегментации изображений с использованием одной и той же сверточной основы дает в результате модель, которая лучше решает обе задачи. Это вполне объяснимо: в малосвязанных задачах всегда какая-то часть информации является общей, в результате объединенная модель получает доступ к большему объему информации о каждой отдельной задаче, нежели модель, обучаемая для решения какой-то конкретной задачи.

В настоящее время под повторным использованием моделей в разных задачах подразумевается использование обученных весов моделей, выполняющих универсальные функции, такие как выделение визуальных признаков. Пример этого вы видели в главе 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), разработчик тут же упаковывает решение в абстрактную библиотеку многократного пользования. Другие программисты, столкнувшись с подобной проблемой в будущем, смогут отыскать существующие библиотеки, загрузить их и использовать в своих проектах. Похожим способом в будущем системы метаобучения смогут собирать новые программы, просеивая глобальную библиотеку высокоуровневых блоков многократного использования. Когда система обнаружит, что ей нужны схожие подпрограммы для нескольких разных задач, она сможет создать абстрактную многократно используемую 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

### 14.5.5 Долгосрочная перспектива

Вот какой я вижу долгосрочную перспективу машинного обучения:

- модели будут больше похожи на программы и 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 способны к более широкому обобщению благодаря богатой алгоритмической природе;

- в частности, модели будут сочетать алгоритмические модули, реализующие возможность формальных рассуждений, поиск и средства абстрагирования с геометрическими модулями, обеспечивающими неформальное знание и распознавание шаблонов. AlphaGo (система, для создания которой потребовалось программное обеспечение, созданное вручную, и множество решений, принятых людьми) являет собой ранний пример того, как может выглядеть такое сочетание символического и геометрического ИИ;
- такие модели будут выращиваться 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. Это приведет к абстракции;
- процесс поиска возможных комбинаций подпрограмм для создания новых моделей будет дискретным процессом поиска (синтез программы), но этот поиск будет в значительной степени проходить направленно благодаря использованию глубокого обучения;
- эта глобальная библиотека и связанная с ней система моделей смогут достичь уровня экстремального обобщения, сопоставимого с человеческим: для новой задачи или ситуации система сможет сконструировать новую работающую модель, используя очень небольшой объем данных, благодаря широте программных примитивов, поддерживающих обобщение, и обширному опыту решения похожих задач. Точно так же люди быстро осваивают новую сложную видеоигру, опираясь на прежний опыт использования других видеоигр, 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, критический анализ этих заблуждений не является целью данной книги.

## 14.6 Как не отстать от прогресса в быстро развивающейся отрасли

В заключение я хочу дать вам несколько советов о том, 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 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 реальное программирование, вот в чем заключается главная идея 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 были выиграны с использованием библиотеки XGBoost (поверхностное машинное обучение) или фреймворка Keras (глубокое обучение). Таким образом, вы вполне подготовлены к участию! Поучаствовав в нескольких состязаниях, возможно, в составе команды, вы познакомитесь с практической стороной некоторых передовых приемов, описанных в этой книге: настройкой гиперпараметров, преодолением проблемы переобучения на проверочном наборе данных и ансамблированием моделей.

### 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 до-

ступны 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 препринт-сервер для размещения статей в области физики, математики и информатики. Он фактически стал основным средоточием ультрасовременных знаний о машинном и глубоком обучении. Подавляющее большинство исследователей глубокого обучения выгружают на сайт arXiv свои статьи, написанные вскоре после состязаний. Это позволяет им поднять флаг и заявить о конкретных находках, не дожидаясь решения конференции (для чего могут потребоваться месяцы), что абсолютно необходимо, учитывая быстрые темпы исследований и высокую конкуренцию в этой области. Это также поддерживает чрезвычайно высокий темп развития области: все новые находки немедленно становятся доступными для всех желающих.

Существенной проблемой является ежедневное появление в arXiv большого количества новых статей, что делает невозможным хотя бы бегло ознакомиться с ними со всеми, а тот факт, что они не подвергаются экспертной оценке, усложняет выявление наиболее важных и ценных из них. С каждым днем становится все труднее выделить полезный сигнал из шума. Но существуют полезные инструменты для отбора статей. В частности, вы можете использовать Google Scholar (<https://scholar.google.com>), чтобы отслеживать публикации ваших любимых авторов.

### 14.6.3 Исследование экосистемы Keras

По состоянию на конец 2021 года у Keras более миллиона пользователей, число которых продолжает расти. Вокруг фреймворка Keras сложилась огромная экосистема из руководств, справочников и проектов с открытым исходным кодом:

- основной справочник по использованию интерфейса R к фреймворку Keras – электронная документация по адресу: <https://keras.rstudio.com> и <https://tensorflow.rstudio.com>. В частности, вы найдете обширные руководства для разработчиков на <http://tensorflow.rstudio.com/guides>, десятки высококачественных примеров кода Keras на <http://tensorflow.rstudio.com/examples> и множество инструкций на <http://tensorflow.rstudio.com/tutorials>. Обязательно изучите их!
- смело обращайтесь к документации Python для Keras и TensorFlow, доступной по адресу [https://www.tensorflow.org/api\\_docs/python/tf](https://www.tensorflow.org/api_docs/python/tf) и <https://keras.io/>, даже если вы не знаете Python. Почти все, что там написано, вы можете понять и применить к интерфейсу R без особого знания языка Python. (Если вы столкнулись с непонятным синтаксисом Python, воспользуйтесь приложением к этой книге);

- исходный код R для Keras и Tensorflow можно найти на сайтах <https://github.com/rstudio/keras> и <https://github.com/rstudio/tensorflow>. Исходные коды Python и C++ доступны по адресу <https://github.com/keras-team/keras> и <https://github.com/tensorflow/tensorflow>. Все исходные коды распространяются по открытой лицензии;
- вы можете обратиться за помощью и присоединиться к обсуждениям глубокого обучения в следующих местах:
  - раздел машинного обучения сообщества Rstudio: <https://community.rstudio.com/c/ml/15>;
  - Stack Overflow: <https://stackoverflow.com> (обязательно пометьте свой вопрос тегами R и Keras);
  - список рассылки (Python) Keras: [keras-users@googlegroups.com](mailto:keras-users@googlegroups.com);
  - вы можете подписаться на меня в Твиттере: @fchollet.

## 14.7 *Заключительное слово*

Вот вы и закончили чтение второго издания книги «Глубокое обучение с R». Надеюсь, вы узнали что-то новое о машинном обучении, глубоком обучении, Keras и, может быть, даже о способности мыслить в целом. Обучение – это пожизненное путешествие, особенно в области ИИ, где неизвестностей гораздо больше, чем определенности. Поэтому продолжайте учиться, задавайте вопросы и занимайтесь исследованиями. Никогда не останавливайтесь. Потому что, даже несмотря на достигнутый прогресс, многие фундаментальные вопросы в ИИ пока не имеют ответа. А многие вопросы даже еще не были правильно сформулированы.

# Приложение

## Введение в Python для пользователей R

---

Возможно, вам захочется разобраться с алгоритмами программ, написанных на языке Python, или даже перенести какие-то них на R. Мы добавили в книгу это краткое приложение, чтобы вы могли приступить к своим задачам как можно быстрее. На самом деле R и Python достаточно похожи, так что можно обойтись без обязательного изучения Python целиком. Мы начинаем с контейнеров, а затем рассмотрим классы, дандер-методы, итераторы, контекст и многое другое!

### П.1 Пробелы

Пробелы имеют большое значение в Python. В R выражения группируются в блок кода с помощью фигурных скобок `{}`. В Python блок кода формируется путем одинакового отступа для операторов. Например, блок кода R может выглядеть так:

```
if (TRUE) {  
  cat("Это первое выражение. \n")  
  cat("Это второе выражение. \n")  
}
```

Эквивалент этого кода на языке Python выглядит так:

```
if True:
    print("Это первое выражение.")
    print("Это второе выражение.")
```

В качестве источника отступов Python одинаково принимает табуляцию или пробелы, но если их смешивать, может возникнуть путаница. Большинство руководств по стилю предлагают использовать только пробелы (а IDE использует их по умолчанию).

## П.2 Типы контейнеров

В R `list()` – это контейнер, который вы можете использовать для организации объектов R. Оператор `list()` в R имеет множество функций, и в Python нет единственного прямого эквивалента. Вместо этого есть, как минимум, четыре различных типа контейнеров Python, о которых вам нужно знать: списки, словари, кортежи и наборы.

### П.2.1 Списки

Списки Python обычно создаются с использованием квадратных скобок `[]`. Встроенная в Python функция `list()` больше похожа на функцию приведения, по духу она ближе к `as.list()` в R. Самое важное, что нужно знать о списках Python, – это то, что они изменяются по месту (т. е. без промежуточного присвоения). В приведенном ниже примере `y` отражает изменения, внесенные в `x`, потому что объект списка, на который ссылаются оба символа, изменяется по месту:

```
x = [1, 2, 3]
y = x
x.append(4)
print("x is", x)
```

← у и x теперь относятся к одному и тому же списку!

```
| x is [1, 2, 3, 4]
print("y is", y)
| y is [1, 2, 3, 4]
```

Существует идиома Python, которая может пригодиться пользователям R, – это дополнение списков с помощью метода `append()`. Дополнение списков в R, как правило, выполняется медленно, и его лучше избегать. Но поскольку списки Python изменяются по месту (и при добавлении элементов не создается полная копия списка), вы можете дополнять списки Python без особого ущерба для быстродействия.

При работе со списками Python удобным синтаксическим трюком является использование символов + и \*. Это операторы *конкатенации* и *репликации*, похожие на c() и гер() в R:

```
x = [1]
x
| [1]
x + x
| [1, 1]
x * 3
| [1, 1, 1]
```

Вы можете обращаться напрямую к элементам списка, используя целочисленные индексы, обрамленные квадратными скобками [], но не забывайте, что в Python индексирование начинается с 0:

```
x = [1, 2, 3]
x[0]
| 1
x[1]
| 2
x[2]
| 3
try:
    x[3]
except Exception as e:
    print(e)
list index out of range
```

При использовании отрицательных индексов элементы отсчитываются с конца контейнера:

```
x = [1, 2, 3]
x[-1]
| 3
x[-2]
| 2
x[-3]
| 1
```

Вы можете получить *срез* (фрагмент) списка, используя двоеточие (:) внутри квадратных скобок. Обратите внимание, что в срезе *не* включается последний элемент, указанный в диапазоне. При желании вы также можете указать шаг:

```
x = [1, 2, 3, 4, 5, 6]
x[0:2] ← Получить элементы
          на позициях 0 и 1, но не 2

| [1, 2]
x[1:] ← Получить элементы
        с позиции 1 до конца списка

| [2, 3, 4, 5, 6]
x[:-2] ← Получить элементы с начала списка
        и до второго элемента от конца

| [1, 2, 3, 4]
x[:] ← Получить все элементы (идиома применяется
      для копирования списков без изменения)

| [1, 2, 3, 4, 5, 6]
x[::2] ← Получить все элементы
        с шагом 2

| [1, 3, 5]
x[1::2] ← Получить все элементы
         с индекса 1 до конца, с шагом 2

| [2, 4, 6]
```

## П.2.2 Кортежи

*Кортежи* ведут себя как списки, за исключением того, что они не изменяемы и не имеют таких же методов изменения на месте, как `append()`. Обычно они строятся с использованием пары круглых скобок (), но это не обязательное условие, и вы можете встретить неявный кортеж, определяемый просто серией выражений, разделенных запятыми. Поскольку круглые скобки также могут использоваться для указания порядка операций в таких выражениях, как  $(x + 3) * 4$ , для определения кортежей длины 1 требуется специальный синтаксис: запятая в конце. Кортежи чаще всего встречаются в функциях, которые принимают переменное количество аргументов:

```
x = (1, 2) ← Кортеж длины 2
type(x)

| <class 'tuple'>

len(x)

| 2

x
```

```
| (1, 2)
x = (1,) ← Кортеж длины 1
type(x)
```

```
| <class 'tuple'>
len(x)
```

```
| 1
x
```

```
| (1,)
x = () ← Кортеж длины 0
print(f"{'type(x) = }; {'len(x) = }; {'x = }") ← Пример интерполированных строковых литералов.
                                                    Вы можете выполнять интерполяцию строк в R
                                                    с помощью Glue::Glue()
```

```
| type(x) = <class 'tuple'>; len(x) = 0; x = ()
x = 1, 2 ← Тоже кортеж
type(x)
```

```
| <class 'tuple'>
len(x)
```

```
| 2
x = 1, ← Не используйте одиночную запятую!
        Это кортеж!
type(x)
```

```
| <class 'tuple'>
len(x)
```

```
| 1
```

## УПАКОВКА И РАСПАКОВКА

Кортежи – это контейнер, поддерживающий семантику *упаковки* и *распаковки* Python. Python допускает присвоение значений нескольким переменным в одном выражении. Этот прием называется *распаковкой*.

Например:

```
x = (1, 2, 3)
a, b, c = x
a
```

```
| 1
b
```

```
| 2
```

с

3

Вы можете выполнить аналогичную распаковку в R, используя `zeallot::`%<-%``.

Распаковка кортежа может происходить в различных контекстах, таких как итерация:

```
xx = (("a", 1),
      ("b", 2))
for x1, x2 in xx:
    print("x1 =", x1)
    print("x2 =", x2)
```

```
x1 = a
x2 = 1
x1 = b
x2 = 2
```

Если вы попытаетесь распаковать контейнер в неправильное количество переменных, Python выдаст ошибку:

```
x = (1, 2, 3)
a, b, c = x ← Успешно
a, b = x ← Ошибка: x содержит слишком много значений для распаковки
```

```
Error in py_call_impl(callable, dots$args, dots$keywords):
↳ ValueError: too many values to unpack (expected 2)
```

```
a, b, c, d = x ← Ошибка: x содержит недостаточно значений для распаковки
```

```
Error in py_call_impl(callable, dots$args, dots$keywords):
↳ ValueError: not enough values to unpack (expected 4, got 3)
```

Можно распаковывать переменное количество аргументов, используя `*` в качестве префикса к символу (мы снова встретим префикс `*`, когда будем говорить о функциях):

```
x = (1, 2, 3)
a, *the_rest = x
a
```

1

```
the_rest
```

```
[2, 3]
```

Вы можете также распаковывать вложенные структуры:

```
x = ((1, 2), (3, 4))
(a, b), (c, d) = x
```

## П.2.3 Словари

Словари больше всего похожи на среды R. Они представляют собой контейнер, в котором вы можете получать элементы по их имени, хотя в Python имя (называемое *ключом* на языке Python) не обязательно должно быть строкой, как в R. Это может быть произвольный объект Python, имеющий метод `hash()` (т. е. практически любой объект Python). Словарь можно создать с помощью пар `{key:value}`. Словари, как и списки Python, можно изменять по месту. Заметим, что `reticulate::r_to_py()` преобразует именованные списки R в словари:

```
d = {"key1": 1,
     "key2": 2}
d2 = d

d
| {'key1': 1, 'key2': 2}
d["key1"]
| 1
d["key3"] = 3
d2 ←————— Преобразование по месту!
| {'key1': 1, 'key2': 2, 'key3': 3}
```

Подобно средам R (и в отличие от именованных списков R), вы не можете использовать целочисленные индексы, чтобы получить элемент в определенной позиции словаря. Словари – это *неупорядоченные* контейнеры (однако начиная с Python 3.7 словари сохраняют порядок вставки элементов):

```
d = {"key1": 1, "key2": 2}
d[1] ←—————
```

Ошибка: целое число «1» не является  
одним из ключей словаря

```
| Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 1
```

Контейнер, который больше всего соответствует семантике именованного списка R, – это `OrderedDict` (<http://mng.bz/7y5m>), но он относительно редко встречается в коде Python, поэтому мы не будем его рассматривать.

## П.2.4 Наборы

Наборы – это контейнер, который можно использовать для эффективного отслеживания уникальных элементов или дедупликации списков. Они строятся с использованием пар `{val1, val2}` (как словарь, но без двоеточия). Наборы можно рассматривать как словарь,

в котором вы используете только ключи. У наборов есть много эффективных методов для операций с подмножествами, таких как `intersection()`, `issubset()`, `union()` и т. д.:

```
s = {1, 2, 3}
type(s)
| <class 'set'>
s
| {1, 2, 3}
s.add(1)
s
| {1, 2, 3}
```

### П.3 Итерация в цикле `for`

Оператор `for` в Python можно использовать для перебора контейнеров любого типа:

```
for x in [1, 2, 3]:
    print(x)
| 1
| 2
| 3
```

R имеет относительно ограниченный набор объектов, которым можно передать `for`. Python, для сравнения, предоставляет интерфейс протокола итератора, используя который, разработчики могут определять пользовательские объекты с настраиваемым поведением, вызываемые при обращении к `for`. (Мы рассмотрим пример определения пользовательской итерации, когда перейдем к классам.) Возможно, вы захотите использовать итерацию Python из R, применяя `reticulate`, поэтому сейчас будет полезно обойтись без синтаксического сахара, чтобы показать, что оператор `for` делает в Python и как вы можете выполнить его вручную. Здесь происходят две вещи: сначала из предоставленного объекта создается итератор. Затем новый объект итератора повторно вызывается с помощью `next()` до тех пор, пока он не будет исчерпан:

```
l = [1, 2, 3]
it = iter(l) ← Создание объекта итератора
it
| <list_iterator object at 0x7f5e30fbd190>
```

Вызываем метод `next()` для итератора, пока он не будет исчерпан:

```
next(it)
| 1
next(it)
| 2
next(it)
| 3
next(it)
| Error in py_call_impl(callable, dots$args, dots$keywords): StopIteration
```

В R для этой же цели вы можете использовать `reticulate`:

```
library(reticulate)
l <- r_to_py(list(1, 2, 3))
it <- as_iterator(l)

iter_next(it)
| 1.0
iter_next(it)
| 2.0
iter_next(it)
| 3.0
iter_next(it, completed = "StopIteration")
| [1] "StopIteration"
```

Применение итератора к словарям требует четкого понимания того, что вы перебираете: ключи, значения или и то, и другое. Методы словаря позволяют указать, что именно мы перебираем:

```
d = {"key1": 1, "key2": 2}
for key in d:
    print(key)
| key1
| key2

for value in d.values():
    print(value)
| 1
| 2
```

```
for key, value in d.items():
    print(key, ":", value)
```

```
key1 : 1
key2 : 2
```

### П.3.1 Включения

*Включения* (comprehensions, редко переводится на русский) – это специальная синтаксическая конструкция, которая позволяет вам создавать контейнер, такой как список или словарь, а также выполнять небольшую операцию или отдельное выражение для каждого элемента. Включения можно рассматривать как специальный вариант `lapply` в R. Например:

```
x = [1, 2, 3]
```

```
l = [element + 100 for element in x]
l
```

```
[101, 102, 103]
```

← Включение элементов в список из `x`, где вы прибавляете 100 к каждому элементу

```
d = {str(element) : element + 100
      for element in x}
```

```
d
```

```
{'1': 101, '2': 102, '3': 103}
```

← Включение элементов в словарь из `x`, где ключ является строкой. Функция `str()` в Python похожа на `as.character()` в R

## П.4. Определение функций с помощью `def`

Функции Python определяются с помощью оператора `def`. Синтаксис для указания аргументов функции и значений аргументов по умолчанию очень похож на R:

```
def my_function(name = "World"):
    print("Hello", name)
```

```
my_function()
```

```
Hello World
```

```
my_function("Friend")
```

```
Hello Friend
```

Эквивалентный фрагмент кода R будет выглядеть так:

```
my_function <- function(name = "World") {
  cat("Hello", name, "\n")
}
```

```
my_function()
```

```
| Hello World
my_function("Friend")
```

```
| Hello Friend
```

В отличие от функций R, последнее значение функции не возвращается автоматически. Python требует явного оператора возврата:

```
def fn():
    1
print(fn())
```

```
| None
```

```
def fn():
    return 1
print(fn())
```

```
| 1
```

**ПРИМЕЧАНИЕ** Для опытных пользователей R отдельно отметим, что в Python нет эквивалента аргумента R `promises`. Значения аргументов функции по умолчанию определяются один раз при построении функции. Если вы определяете функцию Python с изменяемым объектом в качестве значения аргумента по умолчанию (например, список Python), результат может вас удивить!

```
def my_func(x = []):
    x.append("was called")
    print(x)
```

```
my_func()
my_func()
my_func()
```

```
| ['was called']
| ['was called', 'was called']
| ['was called', 'was called', 'was called']
```

Вы также можете определить функции Python, которые принимают переменное количество аргументов, подобно ... (оператор троеточия) в R. Основное отличие заключается в том, что троеточие в R не делает различий между именованными и неименованными аргументами, а Python делает. В Python префикс с одной \* соответствует получению безымянных аргументов, а две \*\* означают получение аргументов с *ключевыми словами*:

```
def my_func(*args, **kwargs):
    print("args =", args)
    print("kwargs =", kwargs)
```

| args – это кортеж  
←  
| kwargs – это словарь  
←

```
my_func(1, 2, 3, a = 4, b = 5, c = 6)
```

```
args = (1, 2, 3)
kwargs = {'a': 4, 'b': 5, 'c': 6}
```

В то время как \* и \*\* в сигнатуре определения функции *упаковывают* аргументы, в вызове функции они *распаковывают* аргументы. Распаковка аргументов в вызове функции эквивалентна использованию `do.call()` в R:

```
def my_func(a, b, c):
    print(a, b, c)
```

```
args = (1, 2, 3)
my_func(*args)
```

```
| 1 2 3
```

```
kwargs = {"a": 1, "b": 2, "c": 3}
my_func(**kwargs)
```

```
| 1 2 3
```

## П.5 Определение классов с помощью `class`

Можно сказать, что в R главной единицей композиции кода является функция, а в Python – класс. Вы можете быть очень продвинутым пользователем R и все равно не использовать R6, эталонные классы или аналогичные эквиваленты R для объектно-ориентированного стиля классов Python.

Однако в Python понимание основ того, как работают объекты класса, является обязательным знанием, потому что классы – это способ организовывать и находить методы в Python (в отличие от подхода R, где все методы находятся путем наследования). К счастью, базовые навыки использования классов в Python легко освоить.

Не пугайтесь, если это ваше первое знакомство с объектно-ориентированным программированием. Мы начнем с создания простого класса Python для демонстрационных целей:

```
class MyClass:
    pass ←———— pass означает отсутствие действий
```

```
MyClass
```

```
| <class '__main__.MyClass'>
```

```
type(MyClass)
```

```
| <class 'type'>
```

```
instance = MyClass()
```

```
instance
| <__main__.MyClass object at 0x7f5e30fc7790>
type(instance)
| <class '__main__.MyClass'>
```

Как и оператор `def`, оператор `class` привязывает новый вызываемый символ `MyClass`. Обратите внимание на строгое соглашение об именах: имена классов обычно имеют вид `CamelCase` («горбатый регистр»), а функции – `snake_case` («ползучий регистр»). После определения `MyClass` вы можете взаимодействовать с ним и видеть, что он имеет тип `'type'`. Вызов `MyClass()` создает новый экземпляр объекта класса, который имеет тип `'MyClass'` (пока не обращайтесь к префиксу `__main__`). Экземпляр выводится на печать вместе с его адресом памяти, что прямо указывает на необходимость управлять многими экземплярами класса и что экземпляр является изменяемым (по умолчанию модифицируется на месте).

В первом примере мы определили пустой `MyClass`, но когда мы его просматриваем, мы видим, что он уже имеет набор атрибутов (метод `dir()` в Python эквивалентен `name()` в R):

```
dir(MyClass)
| ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
|  '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
|  '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
|  '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
|  '__str__', '__subclasshook__', '__weakref__']
```

## П.5.1 Что означают символы подчеркивания?

Python обычно указывает на нечто особенное, заключая имя в двойное подчеркивание, а специальный токен, заключенный в двойное подчеркивание, обычно называется *dunder* (*дандер*, сокращение от *double underscore* – двойное подчеркивание). «Специальный» – это не технический термин; он просто означает, что токен вызывает функцию языка Python. Некоторые дандер-токены – это просто способы, которыми авторы кода могут воспользоваться определенными синтаксическими сахарами; другие значения предоставляются интерпретатором, которые иначе было бы трудно получить; третьи предназначены для расширения языковых интерфейсов (например, протокол итерации); и, наконец, небольшая горстка дандеров действительно сложна для понимания. К счастью, пользователю R, желающему использовать некоторые функции Python через `reticulate`, достаточно знать только несколько простых для понимания дандеров.

Самый распространенный дандер-метод, с которым вы столкнетесь при чтении кода Python, – это `__init__()`. Это функция, которая

вызывается при вызове конструктора класса, то есть при создании экземпляра класса. Она предназначена для инициализации нового экземпляра класса. В очень сложных случаях вы также можете встретить классы, в которых также определена функция `__new__()`; она вызывается перед `__init__()`.

```
class MyClass:
    print("MyClass's definition body is being evaluated")

    def __init__(self):
        print(self, "is initializing")

instance = MyClass()

<__main__.MyClass object at 0x7f5e30fcafd0> is initializing
print(instance)

<__main__.MyClass object at 0x7f5e30fcafd0>

instance2 = MyClass()

<__main__.MyClass object at 0x7f5e30fc7790> is initializing
print(instance2)

<__main__.MyClass object at 0x7f5e30fc7790>
```

Определение происходит один раз, когда класс определяется впервые

Обратите внимание на одинаковый адрес памяти `instance` и `self` в методе `__init__()`

Новый экземпляр, новый адрес в памяти

Несколько полезных замечаний:

- оператор класса принимает блок кода, который определяется общим уровнем отступа. Блок кода имеет такую же семантику, как и любое другое выражение, принимающее блок кода, например `if` или `def`. Тело класса просматривается только *один* раз – при первом создании конструктора класса. Помните, что любые объекты, определенные здесь, являются общими для всех экземпляров класса!
- `__init__()` – это обычная функция, определенная с помощью `def`, как и любая другая функция, за исключением того, что она находится внутри тела класса;
- `__init__()` принимает аргумент `self`. Это инициализируемый экземпляр класса (адреса памяти `self` и `instance` совпадают). Заметьте, что мы не предоставили `self` в явном виде при вызове `MyClass()` для создания экземпляра класса – аргумент `self` был неявно добавлен в вызов функции языком;
- `__init__()` вызывается каждый раз при создании нового экземпляра.

Функции, определенные внутри блока кода класса, называются *методами*, и важно знать, что каждый раз, когда они вызываются

из экземпляра класса, экземпляр подключается к вызову функции в качестве первого аргумента. Это относится ко всем функциям, определенным в классе, включая дандеры. Единственное исключение – если функция декорирована чем-то вроде @classmethod или @staticmethod:

```
class MyClass:
    def a_method(self):
        print("MyClass.a_method() was called with", self)

instance = MyClass()
instance.a_method()
```

| MyClass.a\_method() was called with <\_\_main\_\_.MyClass object at 0x7f5e30fcadf0>:

MyClass.a\_method() ←———— Ошибка: отсутствует обязательный аргумент self

| Error in py\_call\_impl(callable, dots\$args, dots\$keywords):

➔ TypeError: a\_method() missing 1 required positional argument: 'self'

MyClass.a\_method(instance) ←———— Идентично instance.a\_method()

| MyClass.a\_method() was called with <\_\_main\_\_.MyClass object at 0x7f5e30fcadf0>

Есть и другие дандеры, о которых стоит знать:

- `__getitem__` – функция, вызываемая при извлечении среза с помощью `[]` (эквивалентно определению метода `[ S3 в R]`);
- `__getattr__` – функция, вызываемая при доступе к атрибуту с помощью оператора точки `.` (эквивалентно определению метода `$ S3 в R]`).
- `__iter__` и `__next__` – функции, вызываемые оператором `for`;
- `__call__` – вызывается при вызове экземпляра класса как функции, например `instance()`;
- `__bool__` – вызывается `if` и `while` (эквивалентно `as.logical()` в R, но возвращает только скаляр, а не вектор);
- `__repr__` и `__str__` – функции, вызываемые для форматирования и красивого вывода (сродни методам `format()`, `dput()` и `print()` в R);
- `__enter__` и `__exit__` – функции, вызываемые оператором `with`;
- многие встроенные функции Python – это просто синтаксический сахар для вызова дандеров. Например, вызов `repr(x)` идентичен `x.__repr__()` (см. <https://docs.python.org/3/library/functions.html>). К встроенным функциям, которые просто служат сахаром для вызова дандеров, также относятся `next()`, `iter()`, `str()`, `list()`, `dict()`, `bool()`, `dir()`, `hash()` и многие другие!

## П.5.2 Еще раз про итераторы

Теперь, когда вы знакомы с основами классов, пришло время вернуться к *итераторам*. Сначала немного терминологии:

- *итерируемый* – объект, метод которого можно повторять. Точнее, класс, определяющий метод итератора, задачей которого является возврат итератора;
- *итератор* – то, что выполняет итерацию. Точнее, класс, определяющий метод `__next__`, задачей которого является возврат следующего элемента при каждом его вызове, а затем инициирование исключения `StopIteration` после его исчерпания. Обычно можно увидеть классы, которые одновременно являются итерируемыми и итераторами, где метод `__iter__` является просто заглушкой, возвращающей `self`. Ниже показана пользовательская реализация итерируемого/итератора `range()` в Python (аналогично `seq()` в R):

```
class MyRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        self._index = self.start - 1  ← Сброс счетчика
        return self

    def __next__(self):
        if self._index < self.end:
            self._index += 1  ← Инкремент на 1
            return self._index
        else:
            raise StopIteration

for x in MyRange(1, 3):
    print(x)
```

```
1
2
3
```

Распишем пошаговые действия оператора `for`:

```
r = MyRange(1, 3)
it = iter(r)
next(it)
```

```
1
next(it)
```

```
2
next(it)
```

```
3
next(it)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): StopIteration
```

## П.6 Определение генераторов с оператором yield

*Генераторы* – это специальные функции Python, которые содержат один или несколько операторов yield. Как только в блоке кода появляется yield, передаваемый в def, его семантика существенно меняется. Теперь вы определяете не простую функцию, а конструктор генератора! В свою очередь, вызов конструктора генератора создает объект генератора, который является просто разновидностью итератора. Вот пример:

```
def my_generator_constructor():
    yield 1
    yield 2
    yield 3
```

На первый взгляд генератор выглядит как обычная функция:

```
my_generator_constructor
```

```
<function my_generator_constructor at 0x7f5e30fab670>
```

```
type(my_generator_constructor)
```

```
<class 'function'>
```

Но вызов генератора возвращает нечто особенное, *объект генератора*:

```
my_generator = my_generator_constructor()
my_generator
```

```
<generator object my_generator_constructor at 0x7f5e3ca52820>
```

```
type(my_generator)
```

```
<class 'generator'>
```

Объект генератора является одновременно итерируемым и итератором. Его метод `__iter__` – просто заглушка, которая возвращает `self`:

```
iter(my_generator) == my_generator == my_generator.__iter__()
```

```
True
```

Его можно пройти по шагам, как и любой другой итератор:

```
next(my_generator)
```

```
1
```

```
my_generator.__next__()
```

```
next(x) is just sugar for calling
the dunder x.__next__()
```

2

```
next(my_generator)
```

3

```
next(my_generator)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): StopIteration
```

Обнаружение `yield` в блоке кода похоже на нажатие кнопки паузы при выполнении функции: этот оператор сохраняет состояние тела функции и возвращает управление тому, что перебирает объект генератора. Вызов `next()` для объекта-генератора возобновляет выполнение тела функции до тех пор, пока не встретится следующий `yield` или функция не завершится. Вы можете создавать генераторы в R с помощью `stop::generator()`.

## П.7 Заключительные замечания по итерации

Итерация глубоко встроена в язык Python, и пользователей R может удивить тот факт, что многие вещи в Python так или иначе имеют отношение к итерации – являются итерируемыми, итераторами или управляются протоколом итератора. Например, встроенная функция `map()` (эквивалентная `lapply()` в R) возвращает итератор, а не список. Точно так же обработка кортежа, например `(elem for elem in x)`, создает итератор. Большинство функций, связанных с файлами, являются итераторами.

Каждый раз, когда вам неудобен итератор, вы можете вынести все элементы в список, используя встроенную функцию Python `list()` или `reticulate::iterate()` в R. Кроме того, если вам нравится читаемость кода с циклами `for`, вы можете реализовать аналогичную семантику в R с помощью `stop::loop()`.

## П.8 Импорт и модули

В R разработчики могут объединять свой код в общие расширения, называемые *пакетами* R, а пользователи R могут получать доступ к объектам из пакетов R с помощью `library()` или `::` (двойное двоеточие). В Python разработчики объединяют код в *модули*, а пользователи получают доступ к модулям с помощью оператора `import`. Рассмотрим строку:

```
import numpy
```

В этой строке кода Python обращается к файловой системе, находит установленный модуль Python с именем `numpy`, загружает его

(читает файл `__init__.py` и создает объект типа модуля), а затем связывает его с именем `numpy`. Ближайший эквивалент в R выглядит так:

```
dplyr <- loadNamespace("dplyr")
```

## П.8.1 Где находятся модули?

В Python расположение файловой системы, в котором выполняется поиск модулей, может быть доступно (и изменено) из списка, найденного в `sys.path`. Это эквивалент Python для `.libPaths()` в R. Файл `sys.path` обычно содержит пути к текущему рабочему каталогу Python, в котором находятся встроенные стандартные библиотеки, модули, установленные администратором, модули, установленные пользователем, значения из переменных среды, такие как `PYTHONPATH`, и любые изменения, внесенные непосредственно в `sys.path` с помощью другого кода в текущем сеансе Python (хотя на практике это встречается относительно редко):

```
import sys
sys.path
```

Текущий каталог обычно находится на пути доступа к модулям

Стандартная библиотека Python и встроенные модули

```
[ '',
  '/home/tomasz/.pyenv/versions/3.9.6/bin',
  '/home/tomasz/.pyenv/versions/3.9.6/lib/python39.zip',
  '/home/tomasz/.pyenv/versions/3.9.6/lib/python3.9',
  '/home/tomasz/.pyenv/versions/3.9.6/lib/python3.9/lib-dynload',
  '/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9/site-packages',
  '/home/tomasz/opt/R-4.1.2/lib/R/site-library/reticulate/python',
  '/home/tomasz/.virtualenvs/r-reticulate/lib/python39.zip',
  '/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9',
  '/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9/lib-dynload']
```

Подмодули reticulate

Дополнительные установленные пакеты Python (например, через pip)

Другие стандартные библиотеки и встроенные функции, на этот раз от virtualenv

Вы можете проверить, откуда был загружен модуль, обратившись к дандеру `__path__` или `__file__` (особенно полезно при устранении неполадок при установке):

```
import os
os.__file__
```

Здесь определяется модуль `os`.  
Это обычный текстовый файл

```
['/home/tomasz/.pyenv/versions/3.9.6/lib/python3.9/os.py']
```

```
numpy.__path__
```

Импортированный нами модуль `numpy` определяется здесь.  
Это каталог с большим количеством файлов

```
['/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9/site-packages/numpy']
```

После загрузки модуля вы можете получить доступ к символам из модуля с помощью оператора точки `.` (эквивалент `::` или `$.environment` в R):

```
numpy.abs(-1)
```

1

Существует также специальный синтаксис для указания символического имени, к которому привязывается модуль при импорте, и для импорта только определенных имен:

```
import numpy
import numpy as np
np is numpy
from numpy import abs
abs is numpy.abs
from numpy import abs as abs2
abs2 is numpy.abs
```

Импорт и привязка к псевдониму 'numpy'

Импорт и привязка к пользовательскому псевдониму 'np'

Тест на идентичность, аналогичный R `dential(np, numpy)`. Возвращает True (истина)

Импорт только `numpy.abs` и привязка к `abs2`

Истина

Импорт только `numpy.abs` и привязка к `abs2`

Истина

Если вам нужен в Python эквивалент `library()` из R, который открывает доступ ко всем именам модулей в пакете, можно воспользоваться подстановочным знаком `*`, хотя так делают относительно редко. Подстановочный знак `*` будет расширен, чтобы охватить все имена в модуле или все символические имена, перечисленные в дандере `__all__`, если он определен:

```
from numpy import *
```

В отличие от R, Python не делает различий между экспортируемыми пакетами и внутренними символическими именами. В Python все имена модулей равны, хотя существует соглашение об именовании, согласно которому имена, предназначенные для использования во внутренних целях, имеют префикс с одним подчеркиванием в начале. (Два начальных символа подчеркивания вызывают расширенную функцию языка «исправление имен», которая выходит за рамки данного введения.)

Если требуется эквивалентный синтаксис R для оператора `import` в Python, вы можете использовать `envir::import_from()` следующим образом:

```
library(envir)

import_from(keras::keras$applications$efficientnet,
            decode_predictions, preprocess_input,
            new_model = EfficientNetB4)

model <- new_model(include_top = TRUE, weights='imagenet')

predictions <- input_data %>%
  preprocess_input() %>%
  predict(model, .) %>%
  decode_predictions()
```

## П.9 Целые числа и числа с плавающей запятой

Пользователей R обычно не заботит разница между целыми числами и числами с плавающей запятой, но в Python это не так. На всякий случай напомним разницу:

- целочисленный тип (`integer`) могут представлять только целые числа, такие как 2 или 3, а не числа с плавающей запятой, такие как 2.3;
- тип с плавающей запятой (`float`) может представлять любое число, но с некоторой степенью неточности.

В R запись простого числового литерала, такого как 3, дает тип с плавающей запятой, тогда как в Python это будет именно целое число. Вы можете создать целочисленный литерал в R, добавив букву `L`, т. е. `3L`. Многие функции Python ожидают аргументы типа `integer` и сообщат об ошибке, если им предоставлено число с плавающей запятой. Допустим, у нас есть функция Python, которая ожидает целое число:

```
def a_strict_Python_function(x):
    assert isinstance(x, int), "x is not an int"
    print("Yay! x was an int")
```

При вызове из R вы должны обязательно передать ей целое число:

```
library(reticulate)
py$a_strict_Python_function(3) ← Ошибка «AssertionError:
                               x не является целым числом»

py$a_strict_Python_function(3L)
py$a_strict_Python_function(as.integer(3)) | Успешно
```

## П.10 Как быть с векторами R?

R – это язык, разработанный в первую очередь для числовых вычислений. Числовые векторы встроены в язык R до такой степени, что он даже не отличает скаляры от векторов. В свою очередь, числовые методы в Python обычно представлены сторонними пакетами (*модулями* на языке Python).

В Python модуль `numpy` чаще всего используется для обработки непрерывных массивов данных. Ближайшим эквивалентом числового вектора R является одномерный массив `NumPy` или, иногда, список скалярных чисел (некоторые знатоки Python могут привести здесь аргументы в пользу `aggau.aggau()`, но этот метод редко встречается в реальном коде Python, и мы не будем его рассматривать).

Массивы `NumPy` очень похожи на тензоры TensorFlow. Например, они имеют одинаковую семантику широковежательной рассылки и очень похожее поведение при индексировании. API `NumPy` обши-

рен, и изучение полного интерфейса NumPy выходит за рамки этого приложения. Тем не менее стоит указать на некоторые потенциальные опасности для пользователей, привыкших к массивам R:

- при индексировании многомерных массивов NumPy последние измерения могут быть опущены и неявно считаются отсутствующими. Следствием этого является то, что итерация по массивам означает итерацию по первому измерению. Например, следующий код перебирает строки матрицы:

```
import numpy as np
m = np.arange(12).reshape((3,4))
m

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

m[0, :] ← Первая строка

array([0, 1, 2, 3])

m[0] ← Тоже первая строка

array([0, 1, 2, 3])

for row in m:
    print(row)

[0 1 2 3]
[4 5 6 7]
[ 8 9 10 11]
```

- многие операции NumPy изменяют массив по месту! Это удивительно для пользователей R (и пользователей TensorFlow), которые привыкли к удобству и безопасности семантики «скопировал–изменил» в R (и TensorFlow). К сожалению, не существует простой схемы или соглашения об именах, на которые можно положиться, чтобы быстро определить, работает ли тот или иной метод «по месту» или создает новую копию массива. Единственный надежный способ – обратиться к документации (<http://mng.bz/mORP>) и провести небольшие эксперименты с `reticulate::repl_python()`.

## П.11 Декораторы

*Декораторы* – это особые функции, которые принимают функцию в качестве аргумента, а затем обычно возвращают другую функцию. Любая функция может быть вызвана как декоратор с помощью символа-оператора @, который является синтаксическим сахаром для этого простого действия:

```
def my_decorator(func):
    func.x = "a decorator modified this function by adding an attribute `x`"
    return func

@my_decorator
def my_function(): pass

def my_function(): pass
my_function = my_decorator(my_function) ← @decorator – это просто
                                         своеобразный синтаксис
```

Декоратор, с которым вы наверняка будете часто сталкиваться, – это `@property`, который автоматически вызывает метод класса при доступе к атрибуту (аналогично `makeActiveBinding()` в R):

```
from datetime import datetime
class MyClass:
    @property
    def a_property(self):
        return f"`a_property` was accessed at {datetime.now().strftime('%X')}"

instance = MyClass()
instance.a_property

| `a_property` was accessed at 10:01:53 AM'
```

Вы можете перенести `@property` из Python в R с помощью `%<-active%` или с помощью `mark_active()`, например таким образом:

```
import_from(glove, glove)
MyClass %py_class% {
  a_property %<-active% function()
    glue("`a_property` was accessed at {format(Sys.time(), '%X')}")
}

instance <- MyClass()
instance$a_property

| [1] "`a_property` was accessed at 10:01:53 AM"

Sys.sleep(1)
instance$a_property

| [1] "`a_property` was accessed at 10:01:54 AM"
```

## П.12 Оператор *with* и контекстное управление

Любой объект, который определяет методы `__enter__` и `__exit__`, реализует «контекстный» протокол и может быть передан с помощью `with`. В качестве примера ниже показана пользовательская реализация менеджера контекста, который временно изменяет текущий рабочий каталог (эквивалентно `withr::with_dir()` в R):

```

from os import getcwd, chdir

class wd_context:
    def __init__(self, wd):
        self.new_wd = wd

    def __enter__(self):
        self.original_wd = getcwd()
        chdir(self.new_wd)

    def __exit__(self, *args):
        chdir(self.original_wd)

getcwd()

| '/home/tomasz/deep-learning-w-R-v2/manuscript'

with wd_context("/tmp"):
    print("in the context, wd is:", getcwd())

| in the context, wd is: /tmp

getcwd()

| '/home/tomasz/deep-learning-w-R-v2/manuscript'

```

← `__exit__` принимает дополнительный аргумент, который часто игнорируется

## П.13 Где узнать больше

Надеемся, что этот краткий справочник по Python послужит хорошей основой для уверенного чтения документации и кода Python, а также для использования модулей Python в коде R через `reticulate`. Конечно, о Python можно узнать гораздо больше. Поиск в Google гарантированно выдает страницы с ответами на вопросы, но не всегда они начинаются с наиболее полезных. Сообщения в блогах и учебные пособия, предназначенные для начинающих, также содержат много полезной информации, но следует отметить, что официальная документация Python превосходна, и я рекомендую начать с нее, если у вас возникнут вопросы:

- <https://docs.Python.org/3/>;
- <https://docs.Python.org/3/library/index.html>.

Встроенный официальный учебник также превосходен и содержит огромный объем информации (но для изучения понадобится много времени): <https://docs.Python.org/3/tutorial/index.html>.

Наконец, не забудьте закрепить полученные знания, проведя небольшие эксперименты с `reticulate::repl_python()`.

Спасибо, что приобрели и прочитали эту книгу!

# Предметный указатель

---

## N

---

N-грамма, 413

## P

---

Python

- включение, 626
- генератор, 633
- дандер (dunder), 629
- декоратор, 638
- итератор, 631
- конкатенация, 619
- кортеж, 620
- метод, 630
- модуль, 634
- набор, 623
- распаковка, 621
- репликация, 619
- словарь, 623
- список, 618
- срез, 620

## A

---

- Абляционное исследование, 337
- Абляция, 337
- Абстракция, 600

Алгоритм

- долгой краткосрочной памяти, 47
- логистической регрессии, 164
- наивный байесовский, 39
- обратного распространения, 34, 88
- случайного леса, 42

Аннотации, 144

Ансамблирование моделей, 557

## Б

---

Безотлагательное выполнение, 121

Бритва Оккама, 206

## В

---

Вариационный автокодировщик, 522

Ввод. См. *Образец*

Вектор, 61

- концептуальный, 524
- смещения, 342

Векторизация текста, 411

Вероятностное моделирование, 39

Временной ряд, 370

Встраивание слов, 435

- позиционное, 460

Выбор элементов, 488

- жадный, 488

стохастический, 489  
 Вывод. См. *Прогноз*  
 Выделение признаков, 306

## Г

---

Гипотеза  
 калейдоскопа, 600  
 многообразия, 182  
 Глубокое обучение, 572  
 Градиент, 81, 83  
 исчезновение, 337  
 Градиентный спуск, 81  
 стохастический, 86  
 Граф  
 вычислений, 89  
 слоев, 249

## Д

---

Данные  
 векторизация, 225  
 разреженные, 429  
 Деревья решений, 42  
 Дистилляция, 32  
 информации, 356  
 Дифференцируемость, 82  
 Долгая краткосрочная память, 387  
 Дообучение, 306, 316  
 Дополнение, 285

## Е

---

Естественный язык, 408  
 обработка, 409

## З

---

Замораживание слоев, 313

## И

---

Иерархия признаков, 335  
 Извлечение признаков, 251  
 Измерение, 61  
 Изображение  
 классификация, 322

сегментация, 322  
 Изоморфизм подграфов, 603  
 Интерполяция, 183  
 Искусственный интеллект, 572

## К

---

Карта  
 активации класса, 364  
 отклика, 282  
 признаков, 282  
 выходная, 282  
 Категория, 56  
 Каузальное заполнение, 478  
 Квантование весов, 235  
 К-кратная перекрестная  
 проверка, 167  
 Класс, 56, 144  
 Классификация, 371  
 бинарная, 144  
 многоклассовая, 144  
 многозначная, 156  
 однозначная, 156  
 с несколькими метками, 144  
 Кодирование  
 категорийное, 158  
 непосредственное, 147  
 позиционное, 459  
 унитарное, 158  
 множественное, 424  
 Кодировщик, 453  
 Коннекционизм, 575  
 Конструирование признаков, 186, 200  
 Контрольные потери, 140

## Л

---

Лексема. См. *Токен*  
 Логистическая регрессия, 39  
 Логический вывод, 140

## М

---

Маскирование, 440  
 Машинное обучение, 572  
 Мера полезности, 179  
 Метка, 56, 144

Метод  
импульса, 87  
опорных векторов, 40  
оптимизации, 87  
скользящего окна, 283  
ядерный, 40  
Метрики, 134  
Мешок  
биграмм, 414  
слов, 414  
триграмм, 414  
Минимум  
глобальный, 88  
локальный, 88  
Многоголовое внимание, 452  
Многообразия, 182  
Модель  
архитектура, 333  
визуализация знаний, 350  
гиперпараметры, 188, 548  
настройщик, 551  
глубина, 31  
емкость, 198  
интерпретируемость, 350  
контрольные точки, 260  
мешка слов, 413  
последовательности, 413, 432  
преобразование, 464  
репрезентативная мощность, 198  
сверточная основа, 306  
топология, 249  
функциональная, 249  
языковая, 487

## Н

---

Наложение рекуррентных слоев, 397  
Начальная подсказка, 493  
Нейронная сеть, 32  
генеративно-состязательная, 522, 534  
начальное состояние, 389  
прямого распространения, 388  
рекуррентная, 372  
двунаправленная, 402  
сверточная, 277  
Нейронное внимание, 446

Нейронный перенос стиля, 512  
Нормализация TF-IDF, 430

## О

---

Обнаружение  
аномалий, 371  
объектов, 322  
событий, 371  
Обобщение, 174  
локальное, 590  
широкое, 593  
экстремальное, 590  
Образец, 56, 144  
Обучение, 80  
генеративное, 267  
глубокое, 28  
машинное, 26  
поверхностное, 32  
прямой проход, 80  
самообучение, 267  
с подкреплением, 267  
цикл, 80  
эпоха, 97  
Обходные связи. См. *Остаточные связи*  
Ограничивающая рамка, 322  
Оптимизатор, 58, 134  
Оптимизация, 83  
модели, 174  
Остаточные связи, 334  
Ось. См. *Измерение*  
Отбор признаков, 179  
Отношение векторов  
геометрическое, 435  
семантическое, 435  
Отсечение весов, 235  
Ошибка выборки, 223  
Ошибка прогноза. См. *Потеря*

## П

---

Пакет (мини-пакет), 144  
Пакетная нормализация, 334, 341  
Параллелизм  
уровень данных, 564  
уровень модели, 564

Параметризация, 33  
 Паскалина, 27  
 Перекрестная энтропия, 151  
 Переменная, 119  
   обучаемая, 121  
 Переменный коэффициент  
 обучения, 520  
 Переобучение, 60  
 Плотная выборка, 186  
 Подготовка данных, 186  
 Потеря, 144  
 Правила, 27  
 Правило короткого пути, 594  
 Представления, 57  
 Преобразование  
   аффинное, 78  
   линейное, 78  
 Признаки  
   конструирование, 41  
 Прогноз, 144  
 Прогнозирование во времени, 371  
 Прогрессивные матрицы Равена, 597  
 Произведение скалярное  
 (тензорное), 72  
 Производная, 82  
 Прореживание, 209  
 Просачивание цели, 224  
 Пространство гипотез, 31, 134, 333  
 Проход  
   обратный, 85  
   прямой, 85

## Р

---

Развертывание, 400  
 Разделяемые свертки, 334  
 Ранняя остановка, 260  
 Распределенное обучение, 564  
 Расширение данных, 289, 301  
 Регрессия, 164  
   векторная, 144  
   скалярная, 144  
 Регуляризация, 187  
   веса, 207  
 Рекуррентное прореживание, 396  
 Релевантность, 451

## С

---

Саккада, 389  
 Самовнимание, 448  
 Свертка  
   с пропусками, 286  
   с разделением по глубине, 344  
   точечная, 344  
   шаг, 284  
   ядро, 283  
 Сегментация  
   маска, 324  
   семантическая, 324  
   экземпляров, 324  
 Синтез программ, 607  
 Скаляр, 61  
 Скорость обучения, 85  
 Слияние шагов, 569  
 Слой, 128  
   весовой коэффициент, 80  
   классификации, 58  
   обучаемые параметры, 80  
   параметры, 33  
   плотно связанный, 128  
   полносвязный, 58  
   рекуррентный, 128  
 Случайная инициализация, 80  
 Смена концепции, 222  
 Статистика встречаемости, 442  
 Стемминг, 412  
 Страйд, 284

## Т

---

Температура softmax, 489  
 Тензор, 60  
   ранг, 62  
   символический, 245  
   срез, 64, 116  
   тип данных, 63  
   трансляция, 70, 117  
   форма, 62  
 Токен, 412  
   маски, 415  
 Токенизация, 412  
 Транспозиция, 75

**У**

---

Униграмма, 424

Утечка

во времени, 192

информации, 188

**Ф**

---

Фактор нелинейности, 150

Функция

общей потери вариации, 518

потерь, 33, 58, 134

восстановления, 527

регуляризации, 527

стоимости, 34

ступенчатая, 391

тензорная, 68

целевая, 34

**Ц**

---

Цель, 144

Цепное правило, 89

**Ч**

---

Частота

документа, 430

термина, 430

**Э**

---

Экспертная система, 26, 37

Экстрактор признаков, 358

Эмоциональная окраска, 148

Книги издательства «ДМК ПРЕСС»  
можно купить оптом и в розницу  
в книготорговой компании «Галактика»  
(представляет интересы издательств  
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38, оф. 10;  
тел.: **(499) 782-38-89**, электронная почта: **books@aliants-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),  
по которому должны быть высланы книги;  
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: <http://www.galaktika-dmk.com/>.

Франсуа Шолле

### **Глубокое обучение с R и Keras**

Главный редактор *Мовчан Д. А.*  
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Перевод *Яценков В. С.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.

Усл. печ. л. 52,49. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)