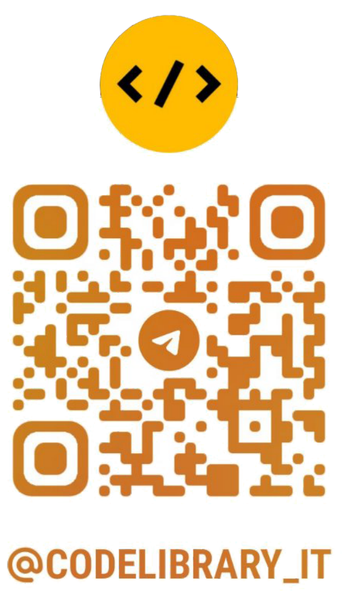
# Генеративное глубокое обучение

## творческий потенциал нейронных сетей

Дэвид Фостер

# Generative Deep Learning

*Teaching Machines to Paint, Write, Compose, and Play*




*David Foster*

Beijing • Boston • Farnham • Sebastopol • Tokyo

# Генеративное глубокое обучение

## творческий потенциал нейронных сетей

Дэвид Фостер

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

2020

ББК 32.818
УДК 004.032.26
Ф81

**Фостер Дэвид**

Ф81 Генеративное глубокое обучение. Творческий потенциал нейронных сетей. — СПб.: Питер, 2020. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1566-2

Генеративное моделирование — одна из самых обсуждаемых тем в области искусственного интеллекта. Машины можно научить рисовать, писать и сочинять музыку. Вы сами можете посадить искусственный интеллект за парту или мольберт, для этого достаточно познакомиться с самыми актуальными примерами генеративных моделей глубокого обучения: вариационные автокодировщики, генеративно-состязательные сети, модели типа кодер-декодер и многое другое.

Дэвид Фостер делает понятными и доступными архитектуру и методы генеративного моделирования, его советы и подсказки сделают ваши модели более творческими и эффективными в обучении. Вы начнете с основ глубокого обучения на базе Keras, а затем перейдете к самым передовым алгоритмам.

- Разберитесь с тем, как вариационные автокодировщики меняют эмоции на фотографиях.
- Создайте сеть GAN с нуля.
- Освойте работу с генеративными моделями генерации текста.
- Узнайте, как генеративные модели помогают агентам выполнять задачи в рамках обучения с подкреплением.
- Изучите BERT, GPT-2, ProGAN, StyleGAN и многое другое.

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

ББК 32.818
УДК 004.032.26

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

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

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

ISBN 978-1492041948 англ.

Authorized Russian translation of the English edition of Generative Deep Learning
© 2019 Applied Data Science Partners Ltd
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1566-2

© Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2020
© Серия «Бестселлеры O'Reilly», 2020

# Краткое содержание

ЧАСТЬ I. ВВЕДЕНИЕ В ГЕНЕРАТИВНОЕ ГЛУБОКОЕ ОБУЧЕНИЕ . . . . . . . 20

Глава 1. Генеративное моделирование . . . . . . . . . . . . . . . . . . . . 21

Глава 2. Глубокое обучение . . . . . . . . . . . . . . . . . . . . . . . . . 56

Глава 3. Вариационные автокодировщики . . . . . . . . . . . . . . . . . . 89

Глава 4. Генеративно-состязательные сети . . . . . . . . . . . . . . . . . 125

ЧАСТЬ II. УЧИМ МАШИНЫ РИСОВАТЬ, ПИСАТЬ, СОЧИНЯТЬ МУЗЫКУ И ИГРАТЬ В ИГРЫ . . . . . . . . . . . . . . . . . . . . . . . . 160

Глава 5. Рисование . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162

Глава 6. Литературное творчество . . . . . . . . . . . . . . . . . . . . . 197

Глава 7. Сочинение музыки . . . . . . . . . . . . . . . . . . . . . . . . . 236

Глава 8. Играем в игры . . . . . . . . . . . . . . . . . . . . . . . . . . . 275

Глава 9. Будущее генеративного моделирования . . . . . . . . . . . . . 316

Глава 10. Заключение . . . . . . . . . . . . . . . . . . . . . . . . . . . . 342

# Оглавление

ПРЕДИСЛОВИЕ . . . . . . . . . . 13
Цели и подходы . . . . . . . . . . 14
Уровень подготовки . . . . . . . . . . 15
Прочие ресурсы . . . . . . . . . . 15
Типографские соглашения . . . . . . . . . . 17
Использование программного кода примеров . . . . . . . . . . 17
Благодарности . . . . . . . . . . 18
От издательства . . . . . . . . . . 19

**ЧАСТЬ I. ВВЕДЕНИЕ В ГЕНЕРАТИВНОЕ ГЛУБОКОЕ ОБУЧЕНИЕ . . . . . . . 20**

**Глава 1. Генеративное моделирование . . . . . . . . . . 21**

Что такое генеративное моделирование? . . . . . . . . . . 21
Генеративное и дискриминативное моделирование . . . . . . . . . . 23
Достижения в машинном обучении . . . . . . . . . . 25
Появление генеративного моделирования . . . . . . . . . . 26
Основа для генеративного моделирования . . . . . . . . . . 29
Вероятностные генеративные модели . . . . . . . . . . 32
Привет, Ирм! . . . . . . . . . . 35
Ваша первая вероятностная генеративная модель . . . . . . . . . . 36
Наивная байесовская параметрическая модель . . . . . . . . . . 41
Привет, Ирм! Продолжение . . . . . . . . . . 44
Сложности генеративного моделирования . . . . . . . . . . 46
Обучение представлению . . . . . . . . . . 48
Настройка окружения . . . . . . . . . . 52
Итоги . . . . . . . . . . 55

**Глава 2. Глубокое обучение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56**

Структурированные и неструктурированные данные . . . . . . . . . . . . 56

Глубокие нейронные сети . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

Keras и TensorFlow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

Ваша первая глубокая нейронная сеть . . . . . . . . . . . . . . . . . . . . 61

Загрузка данных . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

Конструирование модели. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

Компиляция модели . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

Обучение модели . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

Оценка модели . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

Улучшение модели . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

Сверточные слои . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

Пакетная нормализация . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

Слои прореживания. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

Соединяем все вместе . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

**Глава 3. Вариационные автокодировщики . . . . . . . . . . . . . . . . . . 89**

Художественная выставка . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

Автокодировщики . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

Ваш первый автокодировщик . . . . . . . . . . . . . . . . . . . . . . . . . . 94

Кодировщик . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

Декодировщик. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

Объединение кодировщика и декодировщика . . . . . . . . . . . . . . 99

Анализ автокодировщика . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

Выставка вариационного искусства . . . . . . . . . . . . . . . . . . . . . . 104

Конструирование вариационного автокодировщика . . . . . . . . . . . 106

Кодировщик . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

Функция потерь. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

Анализ вариационного автокодировщика. . . . . . . . . . . . . . . . . . 114

Использование вариационного автокодировщика для генерации изображений лиц . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

Обучение VAE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

Анализ вариационного автокодировщика. . . . . . . . . . . . . . . . . . 119

Генерирование новых лиц . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120

Арифметика скрытого пространства. . . . . . . . . . . . . . . . . . . . 121
Преобразование одного лица в другое. . . . . . . . . . . . . . . . . . 123
Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

**Глава 4. Генеративно-состязательные сети . . . . . . . . . . . . . . . . 125**

Ганимал . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
Введение в генеративно-состязательные сети . . . . . . . . . . . . . . . . 128
Ваша первая генеративно-состязательная сеть . . . . . . . . . . . . . . . 129
Дискриминатор . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
Генератор . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
Обучение генеративно-состязательной сети . . . . . . . . . . . . . . 136
Проблемы генеративно-состязательных сетей . . . . . . . . . . . . . . . . 142
Колебания потерь . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
Коллапс модели . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
Неинформативные потери . . . . . . . . . . . . . . . . . . . . . . . . . 144
Гиперпараметры . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
Решение проблем генеративно-состязательных сетей . . . . . . 145
Генеративно-состязательные сети с функцией
потерь Вассерштейна . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
Функция потерь Вассерштейна . . . . . . . . . . . . . . . . . . . . . . 146
Ограничение Липшица . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
Усечение весов . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
Обучение WGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150
Анализ WGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
WGAN-GP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152
Функция потерь штрафа за градиент . . . . . . . . . . . . . . . . . . 153
Анализ WGAN-GP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159

**ЧАСТЬ II. УЧИМ МАШИНЫ РИСОВАТЬ, ПИСАТЬ, СОЧИНЯТЬ МУЗЫКУ И ИГРАТЬ В ИГРЫ . . . . . . . . . . . . . . . . . . . . . . . . 160**

**Глава 5. Рисование . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162**

Яблоки и апельсины . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
CycleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166

Ваша первая сеть CycleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168
Обзор . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168
Генераторы (U-Net) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
Дискриминаторы . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
Компиляция CycleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . 176
Обучение CycleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178
Анализ CycleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179
CycleGAN, рисующая в стиле Моне . . . . . . . . . . . . . . . . . . . . . . . 181
Генераторы (ResNet) . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
Анализ CycleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184
Нейронный перенос стиля . . . . . . . . . . . . . . . . . . . . . . . . . . . 185
Потеря содержимого . . . . . . . . . . . . . . . . . . . . . . . . . . . 187
Потеря стиля. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
Потеря общей дисперсии. . . . . . . . . . . . . . . . . . . . . . . . . 193
Запуск нейронного переноса стиля . . . . . . . . . . . . . . . . . . . 194
Анализ модели нейронного переноса стиля . . . . . . . . . . . . . . 195
Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196

**Глава 6. Литературное творчество . . . . . . . . . . . . . . . . . . . . . . . . 197**

Литературное общество для проблемных правонарушителей . . . . . 198
Сети с долгой краткосрочной памятью . . . . . . . . . . . . . . . . . . . . 200
Ваша первая сеть LSTM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 201
Лексемизация . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 201
Создание набора данных. . . . . . . . . . . . . . . . . . . . . . . . . . 204
Архитектура модели LSTM . . . . . . . . . . . . . . . . . . . . . . . . 205
Слой Embedding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206
Слой LSTM. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 207
Ячейка LSTM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 209
Генерирование нового текста . . . . . . . . . . . . . . . . . . . . . . . . . . 212
Расширения RNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 216
Многослойные рекуррентные сети . . . . . . . . . . . . . . . . . . . 217
Управляемые рекуррентные блоки. . . . . . . . . . . . . . . . . . . . 218
Двунаправленные ячейки . . . . . . . . . . . . . . . . . . . . . . . . . 220
Модели кодировщик-декодировщик. . . . . . . . . . . . . . . . . . . 220
Генератор вопросов и ответов. . . . . . . . . . . . . . . . . . . . . . . 223

Набор данных с вопросами и ответами . . . . . . . . . . . . . . . . . . 224
Архитектура модели . . . . . . . . . . . . . . . . . . . . . . . . . . . . 226
Вычисление результатов . . . . . . . . . . . . . . . . . . . . . . . . . . 231
Результаты моделирования . . . . . . . . . . . . . . . . . . . . . . . . . 233
Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 235

**Глава 7. Сочинение музыки . . . . . . . . . . . . . . . . . . . . . . . . . 236**

Вступление . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 237
Нотная запись . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 237
Ваша первая сеть RNN для генерирования музыки . . . . . . . . . . . . . 240
Внимание . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 242
Конструирование механизма внимания с помощью Keras . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 244
Анализ сети RNN с механизмом внимания . . . . . . . . . . . . . . . 249
Механизм внимания в сетях типа кодировщик-декодировщик . . . . . . . . . . . . . . . . . . . . . . . . . . 254
Генерирование полифонической музыки . . . . . . . . . . . . . . . . 258
Музыкальный орган . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 258
Ваша первая сеть MuseGAN . . . . . . . . . . . . . . . . . . . . . . . . . 260
Генератор MuseGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 263
Аккорды, стиль, мелодия и дорожки . . . . . . . . . . . . . . . . . . 265
Генератор тактов . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 267
Объединяем все вместе . . . . . . . . . . . . . . . . . . . . . . . . . . 268
Критик . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 270
Анализ сети MuseGAN . . . . . . . . . . . . . . . . . . . . . . . . . . 271
Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 273

**Глава 8. Играем в игры . . . . . . . . . . . . . . . . . . . . . . . . . . . 275**

Обучение с подкреплением . . . . . . . . . . . . . . . . . . . . . . . . . 276
OpenAI Gym . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 278
Архитектура модели мира . . . . . . . . . . . . . . . . . . . . . . . . . 280
Вариационный автокодировщик . . . . . . . . . . . . . . . . . . . . . . 281
Сеть MDN-RNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 282
Контроллер . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 283
Подготовка . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 284

Обзор процесса обучения . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 285
Сбор данных в ходе случайных прогонов . . . . . . . . . . . . . . . . . . . . 286
Обучение VAE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 290
Архитектура VAE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 291
Анализ VAE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 291
Сбор данных для обучения RNN . . . . . . . . . . . . . . . . . . . . . . . . . 296
Обучение сети MDN-RNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 297
Архитектура сети MDN-RNN . . . . . . . . . . . . . . . . . . . . . . . . . 298
Выборка следующего состояния и вознаграждения из MDN-RNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 300
Функция потерь в MDN-RNN . . . . . . . . . . . . . . . . . . . . . . . . 300
Обучение контроллера . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 302
Архитектура контроллера . . . . . . . . . . . . . . . . . . . . . . . . . . . 303
CMA-ES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 304
Параллельное выполнение алгоритма CMA-ES . . . . . . . . . . . . 307
Вывод контроллера в процессе обучения. . . . . . . . . . . . . . . . . 309
Обучение в мнимом окружении . . . . . . . . . . . . . . . . . . . . . . . . . 310
Обучение контроллера в мнимом окружении . . . . . . . . . . . . . 312
Недостатки обучения в мнимом окружении . . . . . . . . . . . . . . 314
Итоги . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 315

**Глава 9. Будущее генеративного моделирования . . . . . . . . . . . 316**

Пять лет прогресса . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 317
Трансформер . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 319
Позиционное кодирование. . . . . . . . . . . . . . . . . . . . . . . . . . . 319
Многоголовое внимание . . . . . . . . . . . . . . . . . . . . . . . . . . . . 322
Декодировщик. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 324
Анализ трансформера . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 325
BERT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 326
GPT-2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 327
MuseNet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 328
Достижения в генерировании изображений . . . . . . . . . . . . . . . . 329
ProGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 329
Self-Attention GAN (SAGAN). . . . . . . . . . . . . . . . . . . . . . . . . 332
BigGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 333
StyleGAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 335

Области применения генеративного моделирования . . . . . . . . . . . 339
Изобразительное творчество искусственного интеллекта . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 339
Музыкальное творчество искусственного интеллекта . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 340

**Глава 10. Заключение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 342**

ОБ АВТОРЕ. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 345

ОБ ОБЛОЖКЕ. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 346

# Предисловие

Чего не могу воссоздать, того не понимаю.

*Ричард Фейнман*

Способность творить — неотъемлемая часть человеческой природы. Еще на заре своего существования, живя в пещерах, человек искал возможность создавать оригинальные и красивые творения. Творчество первобытного человека выражалось в наскальных рисунках с дикими животными и абстрактными узорами, созданных с помощью пигментов, аккуратно и методично нанесенных на скалу. Эпоха романтизма подарила нам чудо симфоний Чайковского, с их способностью вызывать чувства триумфа и трагедии посредством звуковых волн, сплетенных в прекрасные мелодии и гармонии. И в наши дни мы, бывает, выскакиваем в полночь из дому, чтобы забежать в книжный магазин и купить продолжение истории о вымышленном волшебнике, потому что комбинация букв образует захватывающее повествование, заставляющее нас переворачивать страницу за страницей, чтобы узнать, какие еще приключения ждут нашего героя.

Поэтому неудивительно, что человечество задалось главным вопросом: можем ли мы создать что-то, что было бы творческим само по себе?

Поиск ответа на этот вопрос является целью генеративного моделирования. Благодаря последним достижениям в науке и технике мы можем создавать машины, способные рисовать оригинальные картины в определенном стиле, писать абзацы связного текста с хорошо прослеживаемой структурой, сочинять музыку, которую приятно слушать, и разрабатывать выигрышные стратегии для сложных игр, генерируя сценарии возможного развития событий. И это только начало генеративной революции, не оставляющей нам другого выбора, кроме отыскания ответов на некоторые из самых важных вопросов о механике творчества и, в конечном итоге, о том, что значит быть человеком.

Иными словами, сейчас самое время заняться изучением генеративного моделирования, так давайте приступим!

## Цели и подходы

В этой книге рассматриваются ключевые методы, доминировавшие в ландшафте генеративного моделирования в последние годы и позволившие добиться впечатляющего прогресса в творческих задачах. Кроме знакомства с базовой теорией генеративного моделирования, в этой книге мы будем создавать действующие примеры некоторых ключевых моделей, заимствованных из литературы, и шаг за шагом рассмотрим реализацию каждой из них.

На протяжении всей книги вам будут встречаться короткие поучительные истории, объясняющие механику некоторых моделей. Пожалуй, один из лучших способов изучения новой абстрактной теории — сначала преобразовать ее во что-то менее абстрактное, например в рассказ, и только потом погружаться в техническое описание. Отдельные разделы теории будут более понятны в контексте, включающем людей, действия и эмоции, а не в контексте таких довольно абстрактных понятий, как, допустим, нейронные сети, обратное распространение или функции потерь.

Рассказ и описание модели — это обычный прием объяснения одного и того же с двух точек зрения. Поэтому, изучая какую-то модель, иногда будет полезно вернуться к соответствующему рассказу. Если же вы уже знакомы с конкретным приемом, то просто получайте удовольствие, обнаруживая в рассказах параллели с каждым элементом модели!

В первой части книги представлены ключевые методы построения генеративных моделей, включая обзор глубокого обучения, вариационных автокодировщиков и генеративно-состязательных сетей. Во второй части эти методы применяются для решения нескольких творческих задач (рисование, сочинение рассказов и музыки) с помощью таких моделей, как CycleGAN, моделей типа кодер-декодер и MuseGAN. Мы увидим, как генеративное моделирование можно использовать для оптимизации выигрышной стратегии игры (World Models), рассмотрим самые передовые генеративные архитектуры, доступные сегодня: StyleGAN, BigGAN, BERT, GPT-2 и MuseNet.

## Уровень подготовки

Эта книга предполагает, что у читателя есть опыт программирования на Python. Если вы не знакомы с Python, начните его изучение с сайта **LearningPython.org** (https://www.learnpython.org/). В интернете есть много бесплатных ресурсов, позволяющих приобрести достаточный объем знаний о Python для работы с примерами из этой книги.

Кроме того, некоторые модели описаны с использованием математических обозначений, поэтому будет полезно иметь достаточно полное представление о линейной алгебре (например, как выполняется умножение матриц и т. д.) и общей теории вероятностей.

Наконец, для опробования примеров кода из репозитория GitHub (https://github.com/davidADSP/GDL_code) книги вам потребуется программное окружение. Примеры для этой книги сознательно создавались так, чтобы не задействовались чрезмерно большие вычислительные ресурсы.

Многие ошибочно считают, что для обучения моделей глубокого обучения необходим графический процессор. Конечно, его наличие не будет лишним и поможет ускорить обучение моделей, но это совсем не обязательно. На самом деле, если вы делаете лишь первые шаги в области глубокого обучения, то начинайте с основ, экспериментируйте с небольшими примерами на ноутбуке и только потом тратьтесь на оборудование для ускорения процесса обучения своих моделей.

## Прочие ресурсы

В качестве общего введения в машинное и глубокое обучение рекомендуются две книги:

- «Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems» Жерона Орельена (Geron Aurelien), вышедшая в издательстве O'Reilly;[1]

[1] *Жерон Орельен*. Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow: Концепции, инструменты и техники для создания интеллектуальных систем. М.: Вильямс, 2018. — *Примеч. пер.*

- «Deep Learning with Python» Шолле Франсуа (Francois Chollet), вышедшая в издательстве Manning.[1]

Большинство статей, упоминаемых в этой книге, получены из **arXiv** (https://arxiv.org/), бесплатного репозитория научных статей. Сейчас многие авторы публикуют свои статьи в arXiv до их рецензирования. Обзор последних поступлений в репозиторий — отличный способ быть в курсе самых передовых разработок в области машинного обучения.

На сайте **Papers with Code** (https://paperswithcode.com/) можно найти самые свежие решения различных задач машинного обучения, ссылки на статьи и официальные репозитории GitHub. Это отличный ресурс для любого желающего выяснить, какие современные методы помогают достичь самых высоких результатов.

Наконец, полезным ресурсом для обучения моделей на высокопроизводительном оборудовании является **Google Colaboratory** (https://colab.research.google.com/). Это бесплатное окружение Jupyter Notebook, не требующее настройки и полностью работающее в облаке. Вы можете настроить выполнение своих сценариев на графическом процессоре, предоставляемом бесплатно (до 12 часов работы). Примеры из этой книги необязательно запускать на графическом процессоре, но такая возможность поможет ускорить процесс обучения ваших моделей. В любом случае Colab — отличный способ получить бесплатный доступ к ресурсам графического процессора.

[1] *Шолле Франсуа*. Глубокое обучение на Python. СПб.: Питер, 2018. — *Примеч. пер.*

## Типографские соглашения

В этой книге приняты следующие типографские соглашения.

*Курсив* используется для обозначения новых терминов, имен файлов и расширений имен файлов.

`Моноширинный шрифт` применяется для оформления листингов программ и программных элементов внутри обычного текста: имен переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Этот элемент используется для обозначения примечаний.

## Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу https://github.com/davidADSP/GDL_code.

В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода из этой книги, ограничиваясь использованием в программе лишь нескольких отрывков кода (в противном случае получение разрешения необходимо, см. ниже). Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, то получение разрешения не требуется.

## Благодарности

Позвольте поблагодарить всех, кто помогал автору в работе над этой книгой.

Прежде всего, искреннее спасибо тем, кто нашел время для научного редактирования, в частности: Любе Эллиотт (Luba Elliott), Даррену Ричардсону (Darren Richardson), Эрику Джорджу (Eric George), Крису Шону (Chris Schon), Сигурдуру Скули Сигургейрссону (Sigurður Skúli Sigurgeirsson), Хао-Вен Донгу (Hao-Wen Dong), Дэвиду Ха (David Ha) и Лорне Барклай (Lorna Barclay).

Кроме того, автор выражает огромную благодарность коллегам из Applied Data Science Partners: Россу Витешчаку (Ross Witeszczak), Крису Шону (Chris Schon), Дэниелу Шарпу (Daniel Sharp) и Эми Булл (Amy Bull). Спасибо за вашу благосклонность в течение всего времени работы над книгой, с нетерпением жду новых проектов машинного обучения, реализацией которых в будущем мы будем заниматься вместе! Особое спасибо Россу за веру в меня как в делового партнера — если бы мы не решили начать совместный бизнес, то эта книга, возможно, никогда бы не появилась!

Я также хочу поблагодарить всех, кто когда-либо учил меня математике, — мне очень повезло, что в школе у меня были фантастические преподаватели, которые развили мой интерес к этому предмету и призывали меня продолжить развивать его в университете. Благодарю вас за вашу самоотдачу и за то, что щедро делились своими знаниями со мной.

Огромное спасибо сотрудникам издательства O'Reilly за то, что помогали мне в процессе работы над этой книгой. Особую благодарность хочу выразить Мишель Кронин (Michele Cronin), помогавшей на каждом этапе полезной обратной связью и посылавшей мне дружеские напоминания о необходимости продолжать писать главы! Спасибо также Кэти Тозер (Katie Tozer), Рейчел Хед (Rachel Head) и Мелани Ярбро (Melanie Yarbrough) за подготовку книги к печати и Майку Лоукидесу (Mike Loukides), который первым обратился ко мне с предложением написать книгу. Вы все оказали огромную поддержку этому проекту, и я хочу поблагодарить вас за то, что вы предоставили мне возможность написать книгу о том, что я люблю.

Все время, пока я писал книгу, моя семья оказывала мне всяческую поддержку. Огромное спасибо моей маме, Джиллиан Фостер, за проверку каждой строки текста на наличие опечаток и особенно за то, что научила

меня считать! Твое внимание к деталям чрезвычайно помогло при редактировании этой книги, и я очень благодарен за все возможности, которые предоставили мне ты и папа. Мой папа, Клайв Фостер, научил меня программированию — эта книга полна практических примеров, которые я сумел написать во многом благодаря его терпению в те годы, когда я изучал Бейсик и пытался писать футбольные игры. Мой брат, Роб Фостер, — самый скромный гений из всех гениев, особенно в области лингвистики, но беседы с ним об искусственном интеллекте и будущем машинного обучения на основе текстов оказались на удивление плодотворными. Наконец, я хотел бы поблагодарить мою сестру Нану — неиссякаемый источник вдохновения и радости. Ее любовь к литературе — одна из причин, по которой я решил, что написать книгу было бы захватывающим занятием.

Наконец, я хотел бы поблагодарить мою невесту (совсем скоро она станет моей женой) Лорну Барклай. Она не только тщательно просмотрела каждое слово в этой книге, но и оказывала мне бесконечную поддержку на протяжении всего процесса, заваривала чай, приносила разные вкусняшки и вообще делала все, чтобы превратить эту книгу в лучшее руководство по генеративному моделированию, уделяя огромное внимание деталям и делясь своими глубокими знаниями в области статистики и машинного обучения. Я едва ли смог бы завершить этот проект без тебя и благодарен за то время, которое ты потратила, чтобы помочь мне реорганизовать и расширить части книги, которые нуждались в дополнительном объяснении. Обещаю, что не буду говорить о генеративном моделировании за обеденным столом, по крайней мере, несколько недель после публикации книги.

## От издательства

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

Мы будем рады узнать ваше мнение!

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

# Часть I

# Введение в генеративное глубокое обучение

**Цель первых четырех глав** — познакомить читателей с основными приемами, необходимыми для того, чтобы начать строить генеративные модели глубокого обучения.

В **главе 1** в общих чертах рассмотрена область генеративного моделирования, задачи, которые эта область решает, а также пример простой вероятностной генеративной модели. Здесь же поясняется, почему по мере роста сложности генеративной задачи необходимо применять методы глубокого обучения.

**Глава 2** содержит руководство по инструментам и методам глубокого обучения, овладение которыми необходимо, чтобы начать создавать более сложные генеративные модели. Эта глава скорее практическое руководство по глубокому обучению, чем теоретический анализ. В частности, здесь представлен фреймворк Keras — инструмент для построения и обучения нейронных сетей с самыми передовыми архитектурами, упоминаемыми в современных источниках.

В **главе 3** рассмотрена наша первая генеративная модель глубокого обучения — вариационный автокодировщик, позволяющий не только создавать реалистичные изображения лиц, но и изменять существующие изображения, например, добавляя улыбку или изменяя цвет волос.

**Глава 4** исследует один из самых успешных методов генеративного моделирования последних лет — генеративно-состязательную сеть. Она служит основой для структурирования задач генеративного моделирования и является основной движущей силой большинства современных генеративных моделей. Читатели узнают, как ее настраивать и адаптировать, чтобы постоянно расширять границы возможностей генеративного моделирования.

ГЛАВА 1

# Генеративное моделирование

Эта глава является общим введением в генеративное моделирование. Сначала посмотрим, что подразумевается под названием *генеративная модель* и чем генеративное моделирование отличается от более широко известного *дискриминативного моделирования*. После этого я представлю базовые принципы и познакомлю с основными математическими понятиями, которые позволят нам сформировать обобщенный подход к задачам, требующим генеративного решения.

Затем, опираясь на новые знания, мы создадим наш первый пример генеративной модели (наивную байесовскую модель), который является вероятностным по своей природе. Он позволит нам генерировать новые образцы, находящиеся за пределами обучающего набора данных, а также исследовать причины, почему модели этого типа могут потерпеть неудачу с увеличением размера и сложности пространства возможных творений.

## Что такое генеративное моделирование?

В общих чертах генеративную модель можно определить так:

> **Генеративная модель описывает, как генерируется набор данных, с точки зрения вероятностной модели. Используя эту модель, можно генерировать новые данные.**

Допустим, у нас есть коллекция изображений лошадей и на ее основе нужно построить модель, способную генерировать новые изображения лошадей, которых никогда не существовало, но которые выглядят реальными, по-

тому что модель выучила общие правила, определяющие внешний вид лошади. Эту задачу можно решить с помощью генеративного моделирования. Краткое описание типичного процесса генеративного моделирования показано на рис. 1.1.

Прежде всего, необходим набор данных, состоящий из множества образцов сущности, которую нужно сгенерировать. Этот набор данных называется *обучающим набором*, а один образец данных в наборе называется *наблюдением*.

Каждое наблюдение состоит из множества *признаков* — в задачах генерации изображений роль признаков обычно играют отдельные пикселы. Наша цель — создать модель, способную генерировать новые наборы признаков, которые выглядят так, будто созданы с использованием тех же правил, что и исходные данные. Концептуально генерация изображений — невероятно сложная задача, учитывая огромное количество способов выбора значений для отдельных пикселов и относительно крошечное число вариантов такого их расположения, когда получается изображение, похожее на моделируемый объект.

Генеративная модель также должна быть *вероятностной*, а не *детерминированной*. Если модель просто представляет фиксированные вычисления, например, выбирает среднее значение каждого пиксела в наборе данных, то она не будет генеративной, потому что каждый раз будет давать один и тот же результат. Модель должна включать *стохастический* (случайный) элемент, который влияет на отдельные выборки, генерируемые моделью.

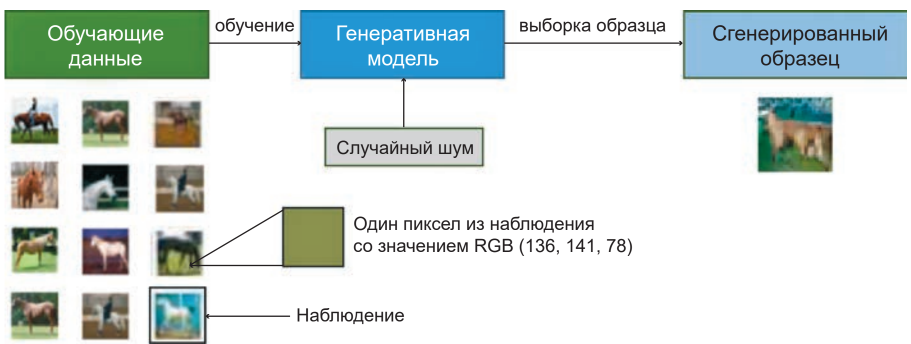



**Рис. 1.1. Процесс генеративного моделирования**

Другими словами, мы можем представить, что существует какое-то неизвестное вероятностное распределение, объясняющее, почему одни изображения могли бы присутствовать в обучающем наборе, а другие нет. Наша задача — создать модель, максимально точно имитирующую это распределение, а затем произвести выборку из нее, чтобы сгенерировать новые наблюдения, которые выглядят так, будто могли бы иметься в исходном обучающем наборе.

## Генеративное и дискриминативное моделирование

Чтобы по-настоящему понять цель и важность генеративного моделирования, полезно сравнить его со своим аналогом, *дискриминативным моделированием*. Знакомые с машинным обучением знают, что большинство задач, с которыми вы столкнетесь, скорее всего, носят дискриминативный характер. Рассмотрим пример, чтобы понять разницу.

Предположим, у нас есть набор данных с коллекцией картин, часть которых написаны Ван Гогом, а часть — другими художниками. Имея достаточный объем данных, мы сможем обучить дискриминативную модель, способную предсказать, была ли данная картина написана Ван Гогом. Наша модель может выучить, какие цвета, формы и текстуры с большей вероятностью будут указывать на принадлежность картины кисти голландского мастера, и в соответствии с этими характеристиками оценивать свой прогноз. На рис. 1.2 показан процесс дискриминативного моделирования — обратите внимание, как он отличается от процесса генеративного моделирования, изображенного на рис. 1.1.

**Рис. 1.2.** Процесс дискриминативного моделирования

Одно из ключевых отличий состоит в том, что при выполнении дискриминативного моделирования каждое наблюдение в обучающих данных имеет *метку*. Для задачи бинарной классификации, такой как определение принадлежности картины, картины Ван Гога будут помечены меткой *1*, а картины других художников — меткой *0*. Исследовав этот набор, наша модель научится различать эти две группы и выведет вероятность, что новое наблюдение имеет метку *1*, то есть что эта картина нарисована Ван Гогом.

По этой причине дискриминативное моделирование часто называют *обучением с учителем* или определением функции отображения входных данных в выходные с использованием маркированного набора данных. Для построения генеративных моделей обычно используются наборы данных без меток (то есть это форма обучения без учителя), хотя также можно использовать наборы данных с метками, чтобы узнать, как генерировать наблюдения из каждого отдельного класса.

Давайте рассмотрим некоторые математические обозначения, помогающие описать разницу между генеративным и дискриминативным моделированием.

> *Дискриминативная модель* оценивает $p(y|\mathbf{x})$ — вероятность метки $y$ для данного наблюдения $\mathbf{x}$.

> *Генеративная модель* оценивает $p(\mathbf{x})$ — вероятность получения наблюдения $\mathbf{x}$.
>
> Если набор данных содержит метки, то можно построить генеративную модель, оценивающую распределение $p(\mathbf{x}|y)$.

Иначе говоря, дискриминативная модель оценивает вероятность того, что наблюдение $\mathbf{x}$ относится к категории $y$. Генеративная модель не учитывает метки наблюдений и оценивает вероятность того, что сгенерированное наблюдение похоже на остальные наблюдения.

Важно отметить, что даже если бы мы были в состоянии создать идеальную дискриминативную модель для идентификации картин Ван Гога, то она все равно не смогла бы создать картину, похожую на картины Ван Гога. Она сможет лишь определять вероятности для существующих изображений, потому

что ее учили именно этому. Нам же нужно обучить генеративную модель, которая сможет генерировать наборы пикселов, с высокой вероятностью принадлежащие исходному набору обучающих данных.

## Достижения в машинном обучении

Чтобы понять, почему генеративное моделирование можно считать следующим рубежом машинного обучения, сначала нужно разобраться, почему дискриминативное моделирование послужило движущей силой для большинства достижений в методологии машинного обучения за последние два десятка лет, как в науке, так и в промышленности.

С академической точки зрения прогресс в дискриминативном моделировании легко проследить, так как есть возможность измерить показатели производительности по определенным задачам классификации и выявить лучшую методологию в своем классе. Оценить генеративные модели труднее, особенно когда оценка качества получаемых результатов в значительной степени субъективна. Поэтому в последние годы большое внимание уделялось обучению дискриминативных моделей для достижения надежности классификации изображений или текста, сравнимой с человеческой или даже превосходящей ее.

Например, в 2012 году произошел важнейший прорыв в классификации изображений, когда команда во главе с Джеффом Хинтоном (Geoff Hinton) из университета в Торонто выиграла конкурс по распознаванию образов ImageNet Large Scale Visual Recognition Challenge (ILSVRC) со своей глубокой сверточной нейронной сетью. По условиям конкурса требовалось классифицировать изображения и отнести их к одной из тысячи категорий, а полученные результаты использовались для сравнения новейших современных методов. Модель глубокого обучения ошиблась в 16 % случаев, оказавшись намного лучше своего ближайшего преследователя с его 26,2 % ошибок. Это вызвало бум в развитии глубокого обучения и повлекло дальнейшее снижение доли ошибок с каждым годом. Победитель конкурса 2015 года впервые добился сверхчеловеческих результатов с долей ошибок 4 %, а современная модель достигла уровня ошибок всего 2 %. Теперь многие считают эту проблему решенной.

Наряду с простотой публикации измеримых результатов в академической среде, дискриминативное моделирование исторически проще применить для решения практических задач, чем генеративное моделирование. При

решении практических задач нас, как правило, не волнует, *как* были сгенерированы данные, нас интересует лишь, как классифицировать или оценить новый образец. Например:

- при изучении спутникового снимка сотрудника министерства обороны будет интересовать только вероятность присутствия на нем вражеских подразделений, а не вероятность того, что это изображение является спутниковым снимком;
- менеджеру по работе с клиентами будет интереснее узнать, какой эмоциональный настрой имеет полученное им электронное письмо, положительный или отрицательный, и он едва ли посчитает полезной генеративную модель, которая может генерировать образцы электронных писем клиентов, которых не существует в действительности;
- врачу полезнее будет знать вероятность наличия признаков глаукомы на данном снимке сетчатки глаза, а не возможность генерировать новые изображения, похожие на снимки задней стенки глазного яблока.

Поскольку большинство решений, имеющих практическое применение, относится к области дискриминативного моделирования, наблюдается рост числа инструментов вида *машинное обучение как услуга* (Machine-Learning-as-a-Service, MLaaS), нацеленных на коммерциализацию дискриминативного моделирования в промышленности и автоматизацию процессов сборки, проверки и мониторинга, являющихся универсальными для почти всех задач дискриминативного моделирования.

## Появление генеративного моделирования

Несмотря на то что прогресс в области машинного обучения до сих пор в значительной степени обеспечивало дискриминативное моделирование, наиболее интересные достижения в этой области в последние 3–5 лет стали результатом применения методов глубокого обучения для задач генеративного моделирования.

В частности, возросло внимание средств массовой информации к таким проектам генеративного моделирования, как StyleGAN от NVIDIA[1], который

---

[1] Теро Каррас (Tero Karras), Самули Лейн (Samuli Laine) и Тимо Айла (Timo Aila), «A Style-Based Generator Architecture for Generative Adversarial Networks», 12 декабря 2018, https://arxiv.org/abs/1812.04948.

способен создавать очень реалистичные изображения человеческих лиц, и языковая модель GPT-2 от OpenAI[1], которая может завершить отрывок текста по короткому вступительному абзацу.

На рис. 1.3 показан поразительный прогресс в области создания фотореалистичных изображений лица, достигнутый с 2014 года[2]. Эти достижения с успехом могут использоваться в таких отраслях, как разработка компьютерных игр и кинематография, и в этих же областях наверняка найдут практическое применение усовершенствованные модели автоматического создания музыки. Еще неизвестно, будем ли мы в обозримом будущем читать новостные статьи или романы, написанные генеративной моделью, но последние достижения в этой области ошеломляют и позволяют надеяться, что этот день когда-нибудь настанет. Однако, несмотря на восхищение от новых достижений, возникают также этические вопросы, связанные с распространением фальшивого контента в интернете, а это означает, что доверять всему, что поступает к нам по общедоступным каналам связи, становится все труднее.

**Рис. 1.3.** Качество изображений лиц, сгенерированных генеративными моделями, значительно улучшилось за последние четыре года[3]

Помимо практического применения (многие варианты которого пока не обнаружены), существуют еще три причины, по которым генеративное моделирование можно считать ключом к гораздо более сложным формам искусственного интеллекта, недоступным для дискриминативного моделирования.

---

[1] Алек Редфорд (Alec Radford) и др., «Language Models Are Unsupervised Multitask Learners», 2019, https://paperswithcode.com/paper/language-models-are-unsupervised-multitask.

[2] Майлз Брандейдж (Miles Brundage) и др., «The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation», февраль 2018, https://www.eff.org/files/2018/02/20/malicious_ai_report_final.pdf.

[3] Источник: Майлз Брандейдж (Miles Brundage) и др., 2018.

Во-первых, с чисто теоретической точки зрения мы не должны довольствоваться успехами в области классификации данных, но должны стремиться к более полному пониманию, как эти данные были созданы. Это, несомненно, более трудная задача из-за высокой размерности пространства возможных выходных данных и относительно небольшого числа творений, которые мы классифицировали бы как принадлежащие набору данных. Однако, как мы увидим далее, многие из методов, которые привели к развитию дискриминативного моделирования, такие как глубокое обучение, могут использоваться и в генеративных моделях.

Во-вторых, весьма вероятно, что генеративное моделирование окажет существенное влияние на выбор направления будущих разработок в других областях машинного обучения, таких как обучение с подкреплением (когда обучаемый агент получает возможность оптимизировать цели методом проб и ошибок). Например, обучение с подкреплением можно использовать, чтобы обучить робота ходить по данной местности. Для этого можно построить компьютерную имитацию местности, а затем провести множество экспериментов, дав агенту возможность опробовать разные стратегии. Со временем агент выявит наиболее успешные стратегии и, следовательно, будет постепенно совершенствоваться. Типичная проблема этого подхода состоит в том, что физика окружающей среды часто очень сложна и ее необходимо рассчитывать на каждом временном шаге, чтобы передать информацию агенту для принятия решения о его следующем шаге. Однако если агент сможет моделировать свое окружение с помощью генеративной модели, то ему не придется проверять стратегию в компьютерной модели или в реальном мире, поскольку он сможет учиться в своем *воображаемом* окружении. В главе 8 мы увидим эту идею в действии, на примере обучения автомобиля максимально быстрому движению по трассе, позволив ему учиться непосредственно на собственной галлюцинации.

Наконец, если мы действительно зададимся целью создать машину, обладающую интеллектом, сравнимым с человеческим, то нам определенно потребуется использовать приемы генеративного моделирования. Один из ярких примеров генеративной модели в природе — человек, читающий эту книгу. Отвлекитесь на минутку и проведите мысленный эксперимент, который поможет вам понять, насколько невероятной генеративной моделью вы являетесь. Вы можете закрыть глаза и представить, как будет выглядеть

слон под любыми возможными углами зрения. Вы можете вообразить ряд возможных концовок вашего любимого телешоу и спланировать свою неделю наперед, прорабатывая разные варианты будущего в своем воображении. Современная нейробиологическая теория предполагает, что наше восприятие реальности — это не сложная дискриминативная модель, получающая информацию от органов чувств и производящая предсказания на ее основе, а генеративная модель, которая с рождения обучается моделированию нашего окружения, точно соответствующего будущему. Некоторые теории даже предполагают, что результатом этой генеративной модели является наше непосредственное восприятие реальности. Очевидно, что глубокое понимание того, как создавать машины, обладающие этими способностями, будет иметь ключевое значение для нашего дальнейшего понимания работы мозга в частности и искусственного интеллекта в целом.

А теперь, после краткого знакомства с историей и перспективами, начнем наше путешествие в захватывающий мир генеративного моделирования. Для начала рассмотрим простейшие примеры генеративных моделей и некоторые идеи, которые помогут нам перейти к более сложным архитектурам, с которыми мы столкнемся позже в книге.

## Основа для генеративного моделирования

Для начала поиграем в генеративное моделирование в двух измерениях. Я выбрал правило, согласно которому был сгенерирован набор точек $\mathbf{X}$, изображенных на рис. 1.4. Назовем это правило $p_{data}$. Ваша задача состоит в том, чтобы выбрать другую точку $\mathbf{x} = (x_1, x_2)$ в пространстве, которая выглядит так, будто она сгенерирована тем же правилом.

Выбрали? Вы, вероятно, использовали свои знания о существующих точках, чтобы построить ментальную модель $p_{model}$ и определить, где в пространстве должна находиться эта точка. В этом отношении $p_{model}$ является *оценкой* $p_{data}$. Возможно, вы решили, что $p_{model}$ должна выглядеть, как показано на рис. 1.5, — прямоугольное поле, в котором могут находиться точки, и область за границами прямоугольника, где точки находиться не должны. Чтобы сгенерировать новое наблюдение, можно просто выбрать случайную точку внутри поля или, выражаясь более формальным языком, *выбрать образец* из распределения $p_{model}$. Поздравляю, вы только что разработали свою первую генеративную модель!

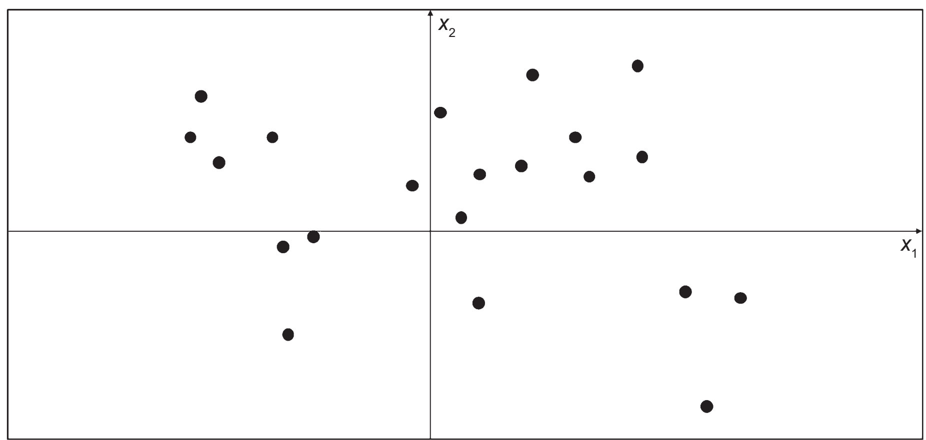



**Рис. 1.4.** Набор точек на двумерной плоскости, сгенерированный с использованием неизвестного правила $p_{data}$

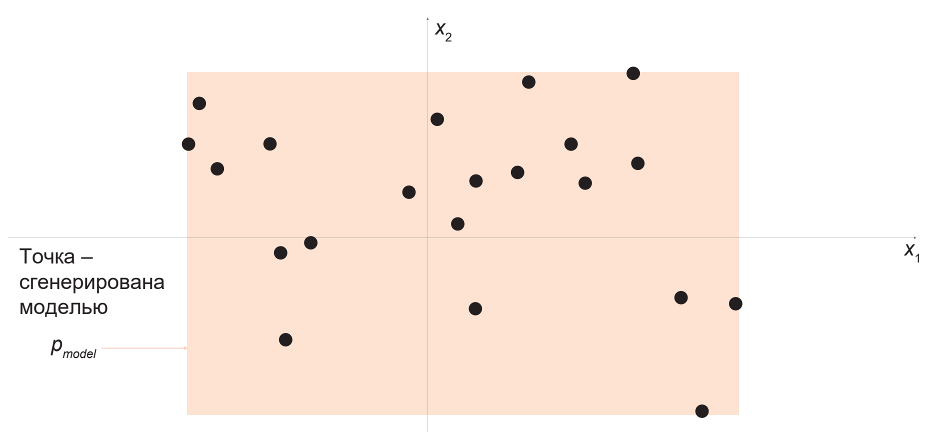



**Рис. 1.5.** Оранжевый прямоугольник $p_{model}$ — это оценка фактического распределения $p_{data}$

Это не самый сложный пример, но его можно использовать, чтобы понять цель генеративного моделирования. Следующие базовые принципы определяют наши мотивы.

Теперь раскроем истинное распределение данных $p_{data}$ и посмотрим, как базовые принципы применяются к этому примеру.

Как показано на рис. 1.6, правило, на основе которого получены данные, — это просто равномерное распределение точек по земной суше.

Очевидно, что наша модель $p_{model}$ — это упрощенное представление $p_{data}$. Точки A, B и C соответствуют трем наблюдениям, сгенерированным моделью $p_{model}$ с разной степенью успеха:

- **Точка A** нарушает правило 1 базовых принципов генеративного моделирования — ясно видно, что она не принадлежит распределению $p_{data}$, поскольку находится посреди моря.
- **Точка B** настолько близко расположена от точки из исходного набора данных, что мы не впечатлены способностью модели создать такую точку. Если все образцы, сгенерированные моделью, будут расположены так же близко к точкам из исходного набора, то это можно считать нарушением правила 2 базовых принципов генеративного моделирования.
- **Точку C** можно рассматривать как успех, потому что она вполне могла быть получена из распределения $p_{data}$ и существенно отличается от всех точек в исходном наборе.

**Рис. 1.6.** Оранжевые прямоугольники $p_{model}$ — это оценка истинного распределения $p_{data}$ (серые области)

**БАЗОВЫЕ ПРИНЦИПЫ ГЕНЕРАТИВНОГО МОДЕЛИРОВАНИЯ**

- У нас есть набор данных с наблюдениями **X**.
- Предполагается, что наблюдения сгенерированы в соответствии с некоторым неизвестным распределением $p_{data}$.
- Генеративная модель $p_{model}$ пытается имитировать $p_{data}$. Правильно подобрав модель $p_{model}$, мы сможем с ее помощью генерировать наблюдения, которые выглядят так, будто были получены из $p_{data}$.
- Модель $p_{model}$ впечатлит нас, если:
  - ✓ Правило 1: она сможет генерировать образцы, которые выглядят так, будто получены из $p_{data}$.
  - ✓ Правило 2: она сможет генерировать образцы, отличающиеся от наблюдений в **X**. То есть модель не должна просто воспроизводить уже известные ей наблюдения.

Область генеративного моделирования разнообразна, и определения задач могут принимать самые разные формы. Однако в большинстве сценариев базовые принципы генеративного моделирования достаточно полно отражают то, насколько широко мы должны думать о решении задачи.

А теперь создадим наш первый пример генеративной модели.

## Вероятностные генеративные модели

Сразу хочу успокоить тех, кто никогда не изучал теорию вероятностей. Чтобы построить и опробовать большинство моделей глубокого обучения, которые мы увидим далее в этой книге, необязательно иметь глубокое понимание математической статистики. Но чтобы получить полное представление об истории задачи, которую мы пытаемся решить, стоит попробовать создать генеративную модель, которая опирается не на глубокое обучение, а исключительно на теорию вероятностей. Это поможет заложить основы для понимания всех генеративных моделей, основанных и не основанных на глубоком обучении, с одной и той же вероятностной точки зрения.

Если вы хорошо знакомы с теорией вероятностей, это здорово, и тогда большая часть следующего раздела будет вам знакома. Но в середине этой главы есть забавный пример, не пропустите его!

Первым делом определим четыре ключевых термина: *выборочное пространство, функция плотности, параметрическое моделирование* и *оценка максимального правдоподобия*.

#### ФУНКЦИЯ ПЛОТНОСТИ ВЕРОЯТНОСТИ

*Функция плотности вероятности* (или просто *функция плотности*), $p(\mathbf{x})$, отображает точку $\mathbf{x}$ из выборочного пространства в число от 0 до 1. Сумма[1] функции плотности по всем точкам в выборочном пространстве должна быть равна 1, то есть это четко определенное распределение вероятностей.[2]

В примере с картой мира функция плотности нашей модели равна 0 за пределами оранжевого прямоугольника и постоянна внутри него.

#### ПАРАМЕТРИЧЕСКОЕ МОДЕЛИРОВАНИЕ

*Параметрическая модель* $p_\theta(\mathbf{x})$ — это семейство функций плотности, которое можно описать с использованием конечного числа параметров $\theta$.

Семейство всех возможных прямоугольников, которые можно было бы нарисовать на рис. 1.5, является примером параметрической модели. В данном случае мы имеем четыре параметра: координаты левого нижнего $(\theta_1, \theta_2)$ и правого верхнего $(\theta_3, \theta_4)$ углов прямоугольника.

Таким образом, каждая функция плотности $p_\theta(\mathbf{x})$ в этой параметрической модели (то есть каждый прямоугольник) может быть уникально представлена четырьмя числами: $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$.

---

[1] Или интеграл, если выборочное пространство непрерывное.

[2] Если выборочное пространство дискретно, тогда $p(\mathbf{x})$ — это просто вероятность принадлежности точки $\mathbf{x}$ к множеству наблюдений.

**ПРАВДОПОДОБИЕ**

Вероятность $\mathcal{L}(\theta|\mathbf{x})$ набора параметров θ является функцией, которая измеряет правдоподобие θ с учетом некоторой наблюдаемой точки **x**.

Определяется она как

$$\mathcal{L}(\theta|\mathbf{x}) = p_\theta(\mathbf{x})$$

То есть правдоподобие θ некоторой наблюдаемой точки **x** определяется как значение функции плотности, параметризованной θ, в точке **x**.

При наличии полного набора данных **X** независимых наблюдений мы могли бы написать:

$$\mathcal{L}(\theta \mid \mathbf{X}) = \prod_{\mathbf{x} \in \mathbf{X}} p_\theta(\mathbf{x})$$

Поскольку с вычислительной точки зрения найти это произведение довольно сложно, вместо него часто используется логарифм правдоподобия ℓ:

$$\ell(\theta \mid \mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} \log p_\theta(\mathbf{x})$$

Есть вполне определенные статистические причины, объясняющие, почему правдоподобие определяется именно так, но нам достаточно будет интуитивного понимания. Мы просто определяем правдоподобие того, что набор параметров θ равен вероятности увидеть данные в рамках модели, параметризованной θ.

В примере с картой мира оранжевый прямоугольник, покрывающий только левую половину карты, имел бы правдоподобие 0 — он не сможет сгенерировать набор данных, потому что мы наблюдаем точки и в правой половине карты. Оранжевый прямоугольник на рис. 1.5 имеет положительное правдоподобие, так как функция плотности положительна для всех точек данных в этой модели.

Существует только одна истинная функция плотности $p_{data}$, которая, как предполагается, сгенерировала наблюдаемый набор данных, и бесконечное число функций плотности $p_{model}$, которые можно использовать для

оценки $p_{data}$. Чтобы структурировать подход к поиску подходящей $p_{model}(\mathbf{X})$, мы можем использовать метод, известный как *параметрическое моделирование*.

Поэтому интуитивно понятно, что цель параметрического моделирования — поиск оптимального значения набора параметров, которое максимизирует вероятность наблюдения набора данных $\mathbf{X}$. Этот метод вполне уместно называют *оценкой максимального правдоподобия*.

**ОЦЕНКА МАКСИМАЛЬНОГО ПРАВДОПОДОБИЯ**

*Оценка максимального правдоподобия* — это метод, позволяющий оценить (набор параметров $\theta$ функции плотности $p_\theta(\mathbf{x})$, которые наиболее вероятно объясняют некоторые наблюдаемые данные $\mathbf{X}$.

Более формально:

$$\hat{\theta} = \arg\max_{\theta} \mathcal{L}(\theta \mid \mathbf{X})$$

$\hat{\theta}$ также называют оценкой максимального правдоподобия (Maximum Likelihood Estimate, MLE).

Теперь, определив все необходимые термины, приступим к описанию порядка создания вероятностной генеративной модели.

В большинстве глав этой книги содержатся короткие рассказы, помогающие описать конкретные приемы. В этой главе мы начнем с путешествия на планету Ирм, где нас ждет первое задание по генеративному моделированию...

## Привет, Ирм!

Сейчас 2047 год, и вас, к вашей радости, назначили новым фэшн-директором на планете Ирм. Теперь вы несете единоличную ответственность за создание новых модных тенденций для жителей планеты.

Ирмяне, как известно, большие модники, поэтому вы должны создавать новые стили, похожие на те, что уже существуют на планете, но не идентичные им.

**Рис. 1.7.** 50 портретов ирмян[1]

По прибытии вам предоставляется набор данных с 50 наблюдениями моды на Ирме (рис. 1.7) и дается один день, чтобы придумать 10 новых стилей и представить их в полицию моды для проверки. Создавая свои шедевры, вы можете экспериментировать с прическами, цветом волос, очками, типом и цветом одежды.

Оставаясь в глубине души специалистом по обработке данных, вы хотите для решения задачи развернуть генеративную модель. После краткого посещения Межгалактической библиотеки вы берете книгу под названием «Генеративное глубокое обучение» и начинаете читать...

*Продолжение следует...*

## Ваша первая вероятностная генеративная модель

Давайте внимательнее рассмотрим набор данных, врученный вам на Ирме. Он состоит из $N = 50$ наблюдений модных стилей, распространенных на планете в настоящее время. Каждое наблюдение можно описать пятью признаками (*тип аксессуаров*, *цвет одежды*, *тип одежды*, *цвет волос*, *прическа*), как показано в табл. 1.1.

[1] Изображения взяты с сайта https://getavataaars.com.

**Таблица 1.1.** Первые 10 наблюдений в наборе данных с портретами ирмян

| № п/п | Аксессуар | Цвет одежды | Тип одежды | Цвет волос | Прическа |
|---|---|---|---|---|---|
| 1 | Круглые очки | Белый | Футболка с круглым вырезом | Рыжий | Короткая стрижка, прямые волосы |
| 2 | Круглые очки | Белый | Комбинезон | Серебристо-серый | Короткая стрижка, курчавые волосы |
| 3 | Солнцезащитные очки | Белый | Футболка с круглым вырезом | Блонд | Короткая стрижка, прямые волосы |
| 4 | Круглые очки | Белый | Футболка с круглым вырезом | Рыжий | Длинная стрижка, прямые волосы |
| 5 | Круглые очки | Белый | Комбинезон | Серебристо-серый | Нет волос |
| 6 | Нет | Белый | Комбинезон | Черный | Длинная стрижка, прямые волосы |
| 7 | Солнцезащитные очки | Белый | Комбинезон | Серебристо-серый | Длинная стрижка, прямые волосы |
| 8 | Круглые очки | Белый | Футболка с круглым вырезом | Серебристо-серый | Короткая стрижка, прямые волосы |
| 9 | Круглые очки | Розовый | Худи | Серебристо-серый | Длинная стрижка, прямые волосы |
| 10 | Круглые очки | Пастельный оранжевый | Футболка с круглым вырезом | Блонд | Длинная стрижка, прямые волосы |

Каждый признак может иметь следующие возможные значения:

- 7 причесок:
  - нет волос;
  - длинные волосы, собранные в пучок;
  - длинная стрижка, волнистые волосы;
  - длинная стрижка, прямые волосы;
  - короткая стрижка, волнистые волосы;

- короткая стрижка, прямые волосы;
- короткая стрижка, курчавые волосы.

- 6 цветов волос:
  - черный;
  - блонд;
  - каштановый;
  - пастельный розовый;
  - рыжий;
  - серебристо-серый.
- 3 вида очков (аксессуаров):
  - нет очков;
  - круглые очки;
  - солнцезащитные очки.
- 4 типа одежды:
  - худи;
  - комбинезон;
  - футболка с круглым вырезом;
  - футболка с V-образным вырезом.
- 8 цветов одежды:
  - черный;
  - синий01;
  - серый01;
  - пастельный зеленый;
  - пастельный оранжевый;
  - розовый;
  - красный;
  - белый.

Итого возможно 7 × 6 × 3 × 4 × 8 = 4032 разные комбинации этих признаков, то есть выборочное пространство насчитывает 4032 точки.

Мы можем представить, что наш набор данных сгенерирован некоторым распределением $p_{data}$, в котором одни значения признаков пользуются большим предпочтением, чем другие. Например, как можно видеть на рис. 1.7, белая одежда, кажется, пользуется большой популярностью, так же как серебристо-серые волосы и футболки с круглым вырезом.

Проблема в том, что истинное распределение $p_{data}$ неизвестно. Нам доступна только выборка наблюдений $\mathbf{X}$, сгенерированных распределением $p_{data}$. Целью генеративного моделирования является построение модели $p_{model}$ с использованием этих наблюдений, которая может точно имитировать наблюдения, производимые $p_{data}$.

Чтобы достичь этой цели, можно просто назначить вероятность каждой возможной комбинации признаков, основываясь на имеющихся данных. Такая параметрическая модель будет иметь $d = 4031$ параметр — по одному на каждую точку в выборочном пространстве — минус один, потому что значение последнего параметра будет выбираться так, чтобы общая сумма была равна 1. Значит, мы должны оценить параметры модели $(\theta_1, ..., \theta_{4031})$.

Этот конкретный класс параметрических моделей известен как *полиномиальное (мультиномиальное) распределение*, а оценка максимального правдоподобия $\hat{\theta}_j$ каждого параметра определяется как

$$\hat{\theta}_j = \frac{n_j}{N},$$

где $n_j$ — число наблюдений с $j$-й комбинацией в наборе данных, а $N = 50$ — общее число наблюдений.

Иначе говоря, оценка каждого параметра — это просто доля от общего количества наблюдений, в которой имела место соответствующая комбинация.

Например, следующая комбинация (назовем ее комбинацией 1) дважды появляется в наборе данных (длинная стрижка, прямые волосы; рыжий; круглые очки; футболка с круглым вырезом; белый).

То есть

$$\hat{\theta}_1 = 2/50 = 0{,}04.$$

Другой пример — комбинация (назовем ее комбинацией 2), которая ни разу не появляется в наборе данных (длинная стрижка, прямые волосы; рыжий; круглые очки; футболка с круглым вырезом; синий01).

То есть

$$\hat{\theta}_2 = 0/50 = 0.$$

Таким способом можно вычислить все значения $\hat{\theta}$ и определить распределение по имеющемуся выборочному пространству. Так как из этого распределения можно выбирать образцы, наш список мог бы называться генеративной моделью. Однако этой модели не хватает главного: она не может генерировать того, что еще не видела, так как для любой комбинации, отсутствующей в исходном наборе данных $\mathbf{X}$, $\hat{\theta}_j = 0$.

Чтобы решить эту проблему, можно назначить дополнительный *псевдоотсчет* от 1 для каждой возможной комбинации признаков. Этот прием известен как *аддитивное сглаживание*, в соответствии с которым оценка максимального правдоподобия (MLE) для параметров будет определяться так:

$$\hat{\theta}_j = \frac{n_j + 1}{N + d}.$$

Теперь каждая комбинация имеет ненулевую вероятность быть выбранной, включая и те, которые отсутствовали в исходном наборе данных, но даже с этим дополнением наша модель не является удовлетворительной генеративной моделью, потому что вероятность наблюдения точки не из исходного набора данных является простой константой. Если попытаться использовать такую модель для создания картин Пикассо, то она бы придала одинаковый вес и случайной коллекции красочных пикселов, и копии картины Пикассо, лишь незначительно отличающейся от настоящей.

В идеале хотелось бы, чтобы наша генеративная модель подчеркивала области выборочного пространства, которые, по ее мнению, более вероятны, согласно некоторой внутренней структуре обучающих данных, а не просто присваивала вероятностные веса точкам, присутствующим в наборе данных.

Для этого нужно выбрать другую параметрическую модель.

## Наивная байесовская параметрическая модель

Наивная байесовская параметрическая модель использует простое предположение, чтобы резко уменьшить количество параметров для оценки. Мы *наивно* предполагаем, что каждый признак $x_j$ *не зависит* от любого другого признака $x_k$.[1] В отношении набора данных, полученных на Ирме, это означает, например, что выбор цвета волос не влияет на выбор типа одежды, а выбор типа очков не влияет на выбор прически. Более формально, для всех признаков $x_j$, $x_k$:

$$p(x_j|x_k) = p(x_j).$$

Это известно как *наивное байесовское* предположение. Чтобы применить его, сначала используется цепное правило вероятности для записи функции плотности в виде произведения условных вероятностей:

$$\begin{aligned} p(\mathbf{x}) = p(x_1, \ldots x_K) = \\ = p(x_2, \ldots x_K \mid x_1) p(x_1) = \\ = p(x_3, \ldots x_K \mid x_2, x_1) p(x_2 \mid x_1) p(x_1) = \\ = \prod_{k=1}^{K} p(x_k \mid x_1, \ldots, x_{k-1}), \end{aligned}$$

где $K$ — общее число признаков (то есть пять в примере с планетой Ирм).

Теперь применим наивное байесовское предположение, чтобы упростить последнюю строку:

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(x_k).$$

Это — наивная байесовская модель. Задача сводится к оценке параметров $\theta_{kl} = p(x_k = l)$ для каждого признака в отдельности и их перемножению для определения вероятности любой возможной комбинации.

Сколько параметров нужно оценить в нашей задаче? Для каждого признака нужно оценить параметр для каждого значения, которое может принять этот

[1] Когда присутствует переменная отклика $y$, наивное байесовское предположение гласит, что существует *условная* независимость между каждой парой признаков $x_j$, $x_k$ при заданном $y$.

признак. Следовательно, в примере с планетой Ирм эта модель определяется всего 7 + 6 + 3 + 4 + 8 – 5 = 23 параметрами.[1]

Оценка максимального правдоподобия $\hat{\theta}_{kl}$ вычисляется как

$$\hat{\theta}_{kl} = \frac{n_{kl}}{N},$$

где $\hat{\theta}_{kl}$ — число раз, когда признак $k$ принимает значение $l$ в наборе данных, а $N = 50$ — общее число наблюдений.

В табл. 1.2 показаны вычисленные параметры для набора данных с планеты Ирм.

Чтобы найти вероятность, с которой модель сгенерирует некоторое наблюдение **x**, достаточно перемножить вероятности отдельных признаков. Например:

$$\begin{gathered} p(\textit{длинная стрижка, прямые волосы; рыжий; круглые очки;} \\ \textit{футболка с круглым вырезом; белый}) = \\ = p(\textit{длинная стрижка, прямые волосы}) \times p(\textit{рыжий}) \times p(\textit{круглые очки}) \times \\ \times p(\textit{футболка с круглым вырезом}) \times p(\textit{белый}) = \\ = 0{,}46 \times 0{,}16 \times 0{,}44 \times 0{,}38 \times 0{,}44 = \\ = 0{,}0054 \end{gathered}$$

Обратите внимание: эта комбинация отсутствует в исходном наборе данных, но наша модель определяет для нее ненулевую вероятность, а значит, вполне может сгенерировать ее. Кроме того, вероятность этой комбинации выше, чем, например, (*длинная стрижка, прямые волосы*; *рыжий*; *круглые очки*; *футболка с круглым вырезом*; *синий01*), потому что белый цвет одежды появляется в наборе наблюдений чаще, чем синий.

То есть наивная байесовская модель способна выявить некоторую структуру данных и использовать ее для создания новых образцов, отсутствующих в исходном наборе. Модель оценила вероятность встретить каждое значение признака независимо от других, поэтому при использовании наивного байесовского предположения можно перемножить эти вероятности, чтобы построить полную функцию плотности, $p_\theta(\mathbf{x})$.

[1] Последний член в выражении –5 отражает тот факт, что последний параметр для каждого признака подбирается так, чтобы сумма его параметров была равна 1.

**Таблица 1.2.** Оценки максимального правдоподобия для параметров в наивной байесовской модели

| **Прическа** | **n** | $\hat{\theta}$ |
|---|---|---|
| нет волос | 7 | 0,14 |
| длинные волосы, собранные в пучок | 0 | 0,00 |
| длинная стрижка, волнистые волосы | 1 | 0,02 |
| длинная стрижка, прямые волосы | 23 | 0,46 |
| короткая стрижка, волнистые волосы | 1 | 0,02 |
| короткая стрижка, прямые волосы | 11 | 0,22 |
| короткая стрижка, курчавые волосы | 7 | 0,14 |
| *Всего* | *50* | *1,00* |

| **Цвет волос** | **n** | $\hat{\theta}$ |
|---|---|---|
| черный | 7 | 0,14 |
| блонд | 6 | 0,12 |
| каштановый | 2 | 0,04 |
| пастельный розовый | 3 | 0,06 |
| рыжий | 8 | 0,16 |
| серебристо-серый | 24 | 0,48 |
| *Всего* | *50* | *1,00* |
| *Всего* | *50* | *1,00* |

| **Цвет одежды** | **n** | $\hat{\theta}$ |
|---|---|---|
| черный | 0 | 0,00 |
| синий01 | 4 | 0,08 |
| серый01 | 10 | 0,20 |
| пастельный зеленый | 5 | 0,10 |
| пастельный оранжевый | 2 | 0,04 |
| розовый | 4 | 0,08 |
| красный | 3 | 0,06 |
| белый | 22 | 0,44 |
| *Всего* | *50* | *1,00* |

| **Вид очков** | **n** | $\hat{\theta}$ |
|---|---|---|
| нет | 11 | 0,22 |
| круглые | 22 | 0,44 |
| солнцезащитные | 17 | 0,34 |
| *Всего* | *50* | *1,00* |

| **Тип одежды** | **n** | $\hat{\theta}$ |
|---|---|---|
| Худи | 7 | 0,14 |
| комбинезон | 18 | 0,36 |
| футболка с круглым вырезом | 19 | 0,38 |
| футболка с V-образным вырезом | 6 | 0,12 |
| *Всего* | *50* | *1,00* |

На рис. 1.8 показаны 10 наблюдений, выбранных моделью.

Для этой простой задачи наивное байесовское предположение о независимости признаков является разумным и, следовательно, дает хорошую генеративную модель.

Теперь посмотрим, что получится, если это предположение оказывается ошибочным.

**Рис. 1.8.** Десять новых стилей для ирмян, сгенерированных наивной байесовской моделью

## Привет, Ирм! Продолжение

Вы испытываете определенное чувство гордости, глядя на десять новых творений, созданных вашей наивной байесовской моделью. Воодушевленные своим успехом, вы обращаете внимание на другую сторону задачи, и на этот раз она не выглядит такой же простой.

Набор данных с незамысловатым названием Planet Pixel, который был вам предоставлен, не содержит пяти высокоуровневых признаков, которые вы видели выше (*цвет волос*, *тип аксессуара* и т. д.), а только значения 32 × 32 пикселов, составляющих каждое изображение. То есть каждое наблюдение теперь имеет 32 × 32 = 1024 признака и каждый признак может принимать любое из 256 значений (отдельные цвета в палитре).

Изображения из нового набора данных показаны на рис. 1.9, а выборка значений пикселов для первых десяти наблюдений показана в табл. 1.3.

Вы решаете попробовать наивную байесовскую модель еще раз, на этот раз обученную на наборе данных пикселов. Модель оценит параметры максимального правдоподобия, определяющие распределение цвета каждого пиксела, чтобы на основе этого распределения сгенерировать новые наблюдения. Однако, закончив модель, вы понимаете, что что-то пошло не так. Вместо новых образцов моды модель вывела десять похожих друг на друга изображений, на которых нельзя различить ни аксессуары, ни четкие признаки прически или одежды (рис. 1.10). Почему так случилось?

**Рис. 1.9.** Образцы моды в Planet Pixel

**Таблица 1.3.** Значения пикселов 458–467 для первых десяти наблюдений в Planet Pixel

| № образца | px_458 | px_459 | px_460 | px_461 | px_462 | px_463 | px_464 | px_465 | px_466 | px_467 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 49 | 14 | 14 | 19 | 7 | 5 | 5 | 12 | 19 | 14 |
| 1 | 43 | 10 | 10 | 17 | 9 | 3 | 3 | 18 | 17 | 10 |
| 2 | 37 | 12 | 12 | 14 | 11 | 4 | 4 | 6 | 14 | 12 |
| 3 | 54 | 9 | 9 | 14 | 10 | 4 | 4 | 16 | 14 | 9 |
| 4 | 2 | 2 | 5 | 2 | 4 | 4 | 4 | 4 | 2 | 5 |
| 5 | 44 | 15 | 15 | 21 | 14 | 3 | 3 | 4 | 21 | 15 |
| 6 | 12 | 9 | 2 | 31 | 16 | 3 | 3 | 16 | 31 | 2 |
| 7 | 36 | 9 | 9 | 13 | 11 | 4 | 4 | 12 | 13 | 9 |
| 8 | 54 | 11 | 11 | 16 | 10 | 4 | 4 | 19 | 16 | 11 |
| 9 | 49 | 17 | 17 | 19 | 12 | 6 | 6 | 22 | 19 | 17 |

**Рис. 1.10.** Десять новых стилей, сгенерированных наивной байесовской моделью на основе набора данных Planet Pixel

## Сложности генеративного моделирования

Во-первых, поскольку наивная байесовская модель выбирает пикселы независимо друг от друга, она не понимает, что два смежных пиксела обычно имеют похожие оттенки, потому что, например, являются частью одного и того же предмета одежды. Модель может сгенерировать цвет лица и рта, поскольку все эти пикселы в обучающем наборе имеют примерно одинаковый оттенок во всех наблюдениях; однако цвет пикселов для футболки выбирается случайно из множества цветов, присутствующих в обучающем наборе, независимо от цвета соседних пикселов. Кроме того, отсутствует механизм формирования пикселов в области вокруг глаз, позволяющий получить круглые очки, или в верхней части изображения, чтобы получить конкретную прическу.

Во-вторых, на этот раз в выборочном пространстве имеется непостижимо большое количество возможных наблюдений. Лишь небольшая часть из них — узнаваемые лица, а еще меньшее подмножество — лица, которые придерживаются правил моды в Planet Pixel. Следовательно, если наивная байесовская модель обучается на сильно коррелированных значениях пикселов, то вероятность найти удовлетворительную комбинацию значений оказывается очень мала.

В первом случае мы имели независимые признаки и относительно небольшое выборочное пространство, поэтому наивная байесовская модель показала неплохие результаты. Во втором случае, когда данные были представлены наборами пикселов, предположение о независимости значений пикселов оказалось несостоятельным. Значения пикселов сильно коррелированы, а выборочное пространство огромно, поэтому получить правильное изображение путем независимой выборки пикселов практически невозможно. Это объясняет, почему наивные байесовские модели плохо работают с необработанными изображениями.

Рассмотренный нами пример высветил две ключевые проблемы, которые должна преодолеть генеративная модель, чтобы добиться успеха.

**СЛОЖНОСТИ ГЕНЕРАТИВНОГО МОДЕЛИРОВАНИЯ**

- Как модель может справиться с высокой условной взаимозависимостью признаков?
- Как модель может отыскать одну из крошечных пропорций, чтобы получить удовлетворительное наблюдение в многомерном выборочном пространстве?

Ключом к решению обеих этих проблем является глубокое обучение.

Нам нужна модель, способная выявить релевантную структуру в данных, которая не требует делать каких-либо предположений заранее. Именно с этим прекрасно справляется глубокое обучение, и именно поэтому этот метод стал движущей силой последних достижений в генеративном моделировании.

Тот факт, что глубокое обучение может формировать свои признаки в пространстве более низкой размерности, означает, что это — форма *обучения представлению*. Прежде чем приступить к глубокому обучению в следующей главе, мы должны разобраться с ключевыми понятиями обучения представлению.

## Обучение представлению

Основная идея обучения представлению заключается в том, чтобы вместо моделирования многомерного выборочного пространства непосредственно попытаться описать каждое наблюдение в обучающем наборе с использованием *некоторого* скрытого пространства меньшей размерности, а затем определить функцию отображения, которая может взять точку из скрытого пространства и отобразить ее в точку в исходном пространстве. Другими словами, каждая точка в скрытом пространстве является *представлением* некоторого многомерного изображения.

Что это означает с практической точки зрения? Предположим, у нас есть обучающий набор, состоящий из черно-белых изображений форм тортов (рис. 1.11).

Совершенно очевидно, что уникально идентифицировать формы тортов можно по двум признакам: высоте и ширине. Зная высоту и ширину, мы сможем нарисовать соответствующую форму, даже если ее изображение отсутствовало в обучающем наборе. Однако для компьютера это не так просто — сначала он должен понять, что высота и ширина являются двумя измерениями скрытого пространства, которые наилучшим образом описывают этот набор данных, а затем сконструировать функцию отображения $f$, которая сможет выбрать точку в этом пространстве и привести ее в соответствие с черно-белым изображением формы. Пример такого скрытого пространства форм тортов и процесс генерации показаны на рис. 1.12.

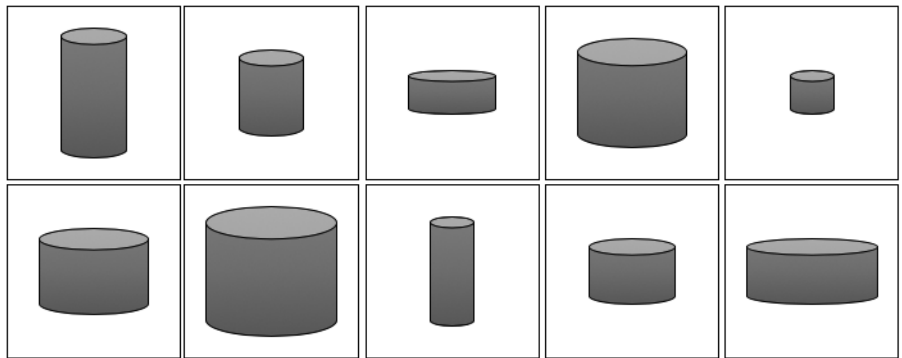


**Рис. 1.11.** Набор изображений форм тортов

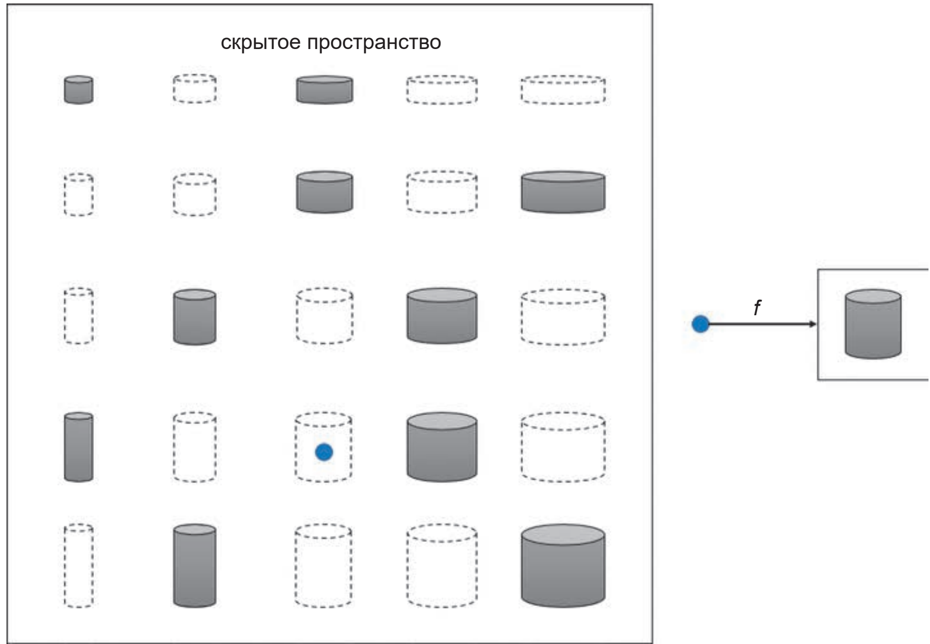



**Рис. 1.12.** Скрытое пространство свойств форм тортов и функция *f*, отображающая точку из скрытого пространства в пространство исходных изображений

Глубокое обучение дает нам возможность разными способами выявлять порой очень сложные функции отображения *f*. Наиболее важные из этих способов мы рассмотрим в последующих главах. А пока достаточно общего понимания, какие цели преследует обучение представлению.

Одно из преимуществ обучения представлению заключается в возможности выполнять операции, влияющие на высокоуровневые свойства изображений, в более управляемом скрытом пространстве. В исходном пространстве изображений неясно, как сгенерировать каждый пиксел, чтобы получить *более высокое* изображение торта. Однако в скрытом пространстве для этого достаточно увеличить на единицу скрытое измерение, отвечающее за *высоту*, а затем применить функцию отображения для перехода обратно в пространство изображений. Пример такого подхода мы рассмотрим в следующей главе, но применительно не к формам тортов, а к граням.

Обучение представлению для нас, людей, настолько естественно, что вы, возможно, никогда не задумывались о том, с какой удивительной легкостью мы делаем это. Представьте, что вы решили описать свою внешность кому-то, кто должен отыскать вас в толпе и не знает, как вы выглядите. Вы едва ли стали бы описывать цвет всех пикселов, формирующих ваше изображение. Вместо этого вы предположили бы, что ваш визави имеет представление о том, как выглядит обычный человек, и заменили описание пикселов высокоуровневыми признаками, соответствующими группам пикселов, например: *у меня очень светлые волосы* или *я ношу очки*. Если вы назовете с десяток таких признаков, то человек сможет отобразить описание обратно в пикселы и создать ваш образ в своей голове. Конечно, образ может получиться далеко не идеальным, но достаточно близким к вашему фактическому внешнему виду, чтобы вас можно было отыскать среди сотен других людей, даже если вас никогда не видели раньше.

Подчеркнем: обучение представлению не просто присваивает значения определенному набору признаков, таких как *светлые волосы*, *рост* и т. д. Сила обучения представлению заключается в том, что этот процесс позволяет выявить в исходных данных наиболее важные признаки, позволяющие описать имеющиеся наблюдения. Говоря математическим языком, цель обучения представлению состоит в том, чтобы найти высоконелинейное *многообразие*, на котором лежат данные, а затем установить измерения, необходимые для полного описания этого пространства (рис. 1.13).

Таким образом, обучение представлению выявляет наиболее важные высокоуровневые признаки, описывающие группы пикселов, поэтому любая точка в скрытом пространстве вполне может представлять правильно сформированное изображение. Изменяя значения признаков в скрытом пространстве, можно создавать новые представления, которые при отображении в пространство исходных изображений имеют гораздо больше шансов выглядеть похожими на *реальные* изображения, чем если бы мы пытались работать с отдельными пикселами.

Познакомившись с идеей обучения представлению, которое является основой многих примеров генеративного глубокого обучения в этой книге, нам остается только настроить окружение, в котором вы сможете начать создавать собственные генеративные модели глубокого обучения.

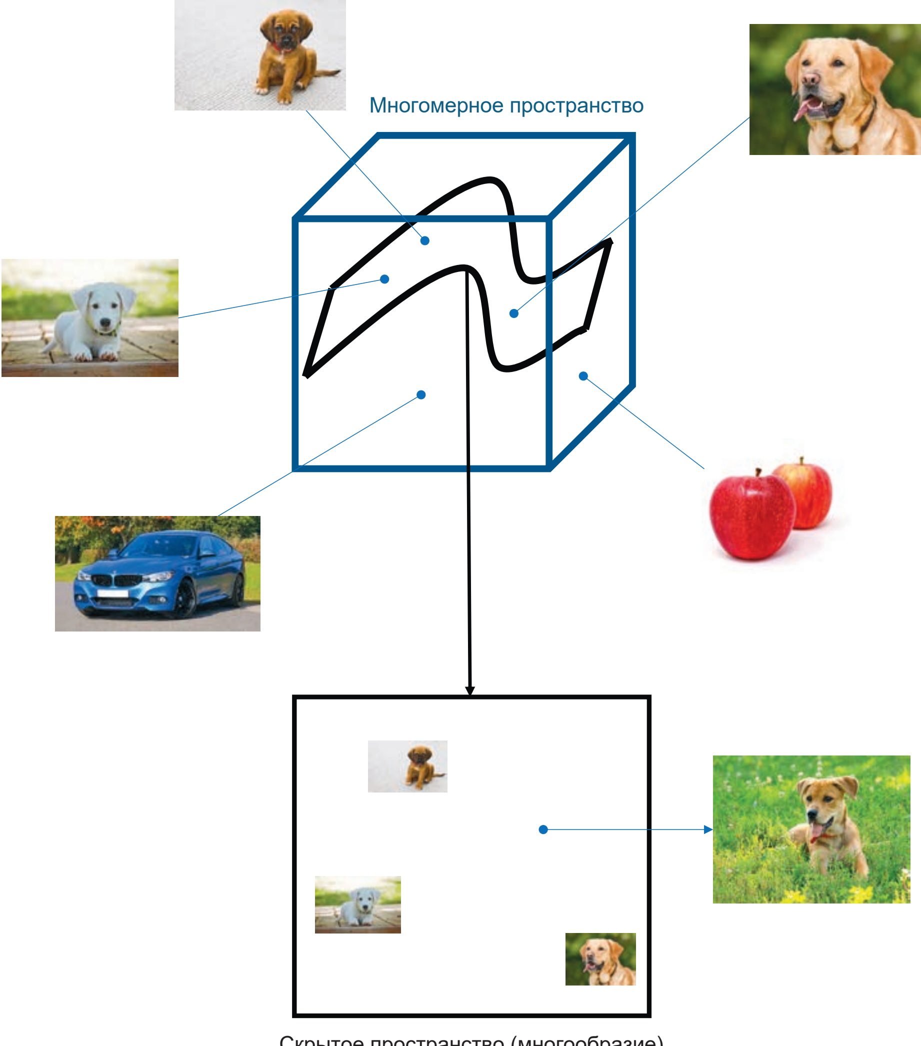



**Рис. 1.13.** Куб представляет многомерное пространство всех изображений; обучение представлению пытается найти скрытое подпространство, или многообразие, с меньшим числом измерений, на котором располагаются определенные виды изображений (например, многообразие собак)

## Настройка окружения

В этой книге приводится множество примеров моделей, которые мы будем обсуждать.

Чтобы получить доступ к этим примерам, вы должны клонировать репозиторий Git, созданный для этой книги. Git — это система с открытым исходным кодом для управления версиями, которая позволит вам скопировать код на свой компьютер, чтобы вы могли запускать его локально или в облачном окружении. Возможно, эта система уже установлена у вас. Если нет, то следуйте инструкциям для вашей операционной системы (http://bit.ly/2MUrvN1).

Чтобы клонировать репозиторий с примерами для этой книги, в окне терминала перейдите в папку, куда вы хотели бы сохранить файлы, и введите следующую команду:

```
git clone https://github.com/davidADSP/GDL_code.git
```

Каждый раз, приступая к опробованию примеров, не упускайте возможность получить последнюю версию кода, выполнив команду

```
git pull
```

Теперь в выбранной вами папке на вашем компьютере должны находиться файлы, скопированные из репозитория.

Далее нужно настроить виртуальное окружение. Это самая обычная папка, куда вы должны установить свежую копию Python и все пакеты, используемые в этой книге. При таком подходе вы будете уверены, что ни одна из библиотек, которые мы используем, не затронет системную версию Python. Те, кто пользуется дистрибутивом Python под названием Anaconda, могут настроить виртуальное окружение командой

```
conda create -n generative python=3.6 ipykernel
```

Остальные могут установить *virtualenv* и *virtualenvwrapper* командой[1]

```
pip install virtualenv virtualenvwrapper
```

[1] Исчерпывающие инструкции по установке *virtualenvwrapper* вы найдете в официальной документации (http://bit.ly/2x8LPQ4).

Также добавьте следующие строки в сценарий запуска командной оболочки (например, в *.bash_profile*):

```
export WORKON_HOME=$HOME/.virtualenvs ❶
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3 ❷
source /usr/local/bin/virtualenvwrapper.sh ❸
```

❶ Папка, где находятся ваши виртуальные окружения.

❷ Версия Python по умолчанию, используемая для создания виртуального окружения, причем это должна быть версия Python 3, а не Python 2.

❸ Команда запуска сценария инициализации *virtualenvwrapper*.

Чтобы создать виртуальное окружение с именем *generative*, просто введите следующую команду в терминале:

```
mkvirtualenv generative
```

Узнать, что вы находитесь внутри виртуального окружения, можно по наличию текста `(generative)` в строке приглашения к вводу.

Теперь можно установить все необходимые пакеты, выполнив следующую команду:

```
pip install -r requirements.txt
```

На протяжении всей книги мы будем использовать Python 3. Файл *requirements.txt* содержит имена и номера версий всех пакетов, которые вам понадобятся для запуска примеров.

Для проверки работоспособности виртуального окружения введите команду `python` в терминале и затем попытайтесь импортировать Keras (библиотеку глубокого обучения, которую мы широко будем использовать в этой книге). Вы должны увидеть приглашение Python 3 с сообщением от библиотеки Keras о том, что она использует библиотеку TensorFlow, как показано на рис. 1.14.

Наконец, нужно убедиться, что ваше виртуальное окружение доступно из блокнотов[1] Jupyter Notebook, находящихся на вашем компьютере. Jupyter — это инструмент интерактивного программирования на Python в окне браузера и отличный способ для воплощения новых идей и обмена кодом с единомышленниками. Большинство примеров в этой книге написаны с использованием Jupyter Notebook.

---

[1] Так называются сценарии в Jupyter Notebook. — *Примеч. пер.*

Для этого выполните следующую команду в окне терминала, внутри виртуального окружения:

```
python -m ipykernel install --user --name generative ❶
```

❶ Эта команда откроет доступ к настроенному вами виртуальному окружению (`generative`) из блокнотов Jupyter Notebook.

Для проверки в том же окне терминала перейдите в папку, куда вы клонировали репозиторий с примерами для книги, и выполните команду

```
jupyter notebook
```

На экране должно появиться окно браузера с содержимым, напоминающим изображение на рис. 1.15. Щелкните мышкой на блокноте, который вы хотели бы запустить, и в раскрывающемся списке Kernel → Change kernel выберите виртуальное окружение generative.

Теперь вы готовы начать создавать генеративные сети глубокого обучения.

```
Python 3.6.5 (default, Oct  6 2018, 09:49:35)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import keras
Using TensorFlow backend.
>>> keras.__version__
'2.2.4'
```

**Рис. 1.14.** Проверка работоспособности окружения

**Рис. 1.15.** Jupyter Notebook

## Итоги

В этой главе вы познакомились с генеративным моделированием, важным разделом машинного обучения, дополняющим хорошо изученную область дискриминативного моделирования. В нашем первом примере генеративной модели мы использовали наивное байесовское допущение, чтобы получить вероятностное распределение, способное представлять внутреннюю структуру данных и генерировать образцы, не входящие в обучающий набор. Мы также увидели, как эта модель может потерпеть неудачу с увеличением сложности генеративной задачи, и проанализировали общие проблемы генеративного моделирования. Наконец, мы бросили первый взгляд на обучение представлению — важную идею, лежащую в основе многих генеративных моделей.

В следующей главе мы приступим к исследованию глубокого обучения и увидим, как использовать библиотеку Keras для построения моделей, способных решать задачи дискриминативного моделирования. Это даст нам необходимую основу для дальнейшего изучения генеративного глубокого обучения в последующих главах.

# ГЛАВА 2

# Глубокое обучение

Начнем с упрощенного определения глубокого обучения:

> **Глубокое обучение — это класс алгоритмов машинного обучения, использующих несколько составных слоев с обрабатывающими модулями для выявления высокоуровневых представлений в неструктурированных данных.**

Чтобы уяснить суть глубокого обучения и его значение для генеративного моделирования, взглянем на определение немного шире. Прежде всего уточним, что подразумевается под «неструктурированными данными» и их противоположностью — «структурированными данными»?

## Структурированные и неструктурированные данные

Многие алгоритмы машинного обучения требуют передачи им *структурированных* данных в табличной форме, где каждое наблюдение-строка описывается признаками-столбцами. Например, возраст человека, его доход и количество посещений веб-сайта за последний месяц — все это признаки, которые могут помочь предсказать, подпишется ли человек на определенную онлайн-услугу в следующем месяце. Структурированную таблицу с этими признаками можно было бы использовать для обучения логистической регрессии, случайного леса или модели XGBoost для предсказания ответа на вопрос, подпишется этот человек (1) или нет (0). Здесь каждый отдельный признак содержит крупицу информации о наблюдении, и, анализируя их, модель сможет узнать, как все эти признаки в совокупности влияют на ответ.

Под *неструктурированными* данными подразумеваются любые данные, не сгруппированные естественным образом в столбцы признаков, например изображения, аудиозаписи и текст. Конечно, изображения имеют пространственную структуру, аудиозаписи — временную, видеозаписи — и пространственную, и временную, но поскольку данные не распределены по столбцам признаков, они считаются неструктурированными (рис. 2.1).

Когда данные не структурированы, отдельные пикселы, частоты или символы не несут почти никакой информации. Например, знание того, что 234-й пиксел в изображении имеет темно-коричневый цвет, никак не помогает определить, что изображено — дом или собака. Точно так же и знание того, что 24-й символ в предложении — это буква *e*, никак не позволяет судить, о чем говорится в тексте — о футболе или о политике. Пикселы (символы) — это лишь мазки на холсте, из которых собраны информативные признаки более высокого уровня: изображение дымохода или слово *нападающий*. Если изобразить дымоход на другой стороне дома, то изображение все равно будет содержать дымоход, но та же информация будет передаваться совершенно другими пикселами. Если слово *нападающий* появится в тексте чуть раньше или чуть позже, то текст все равно будет повествовать о футболе, но ту же информацию будут передавать символы в другой позиции. Детальность данных в сочетании с высокой степенью пространственной зависимости не позволяет рассматривать пиксел (символ) как отдельный информационный признак.

СТРУКТУРИРОВАННЫЕ ДАННЫЕ

| id | age | gender | height (cm) | location |
|---|---|---|---|---|
| 0001 | 54 | M | 186 | London |
| 0002 | 35 | F | 166 | New York |
| 0003 | 62 | F | 170 | Amsterdam |
| 0004 | 23 | M | 164 | London |
| 0005 | 25 | M | 180 | Cairo |
| 0006 | 29 | F | 181 | Beijing |
| 0007 | 46 | M | 172 | Chicago |

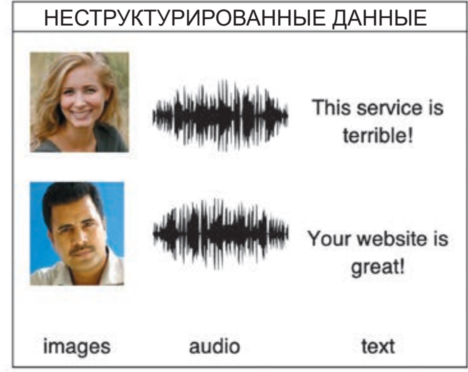



**Рис. 2.1.** Разница между структурированными и неструктурированными данными

Если обучить алгоритм логистической регрессии, случайного леса или XGBoost на исходных значениях пикселов, то обученная модель часто будет работать плохо почти во всех случаях, кроме самых простых задач классификации. Эти модели полагаются на информативность входных признаков и их пространственную зависимость. Модели глубокого обучения, напротив, способны сами выявлять высокоуровневые информативные признаки непосредственно из неструктурированных данных. Глубокое обучение можно применить и к структурированным данным, но его настоящая мощь, особенно в генеративном моделировании, заключается в способности работать с неструктурированными данными. Нам чаще требуется генерировать неструктурированные данные, такие как новые изображения или строки текста, и именно поэтому глубокое обучение оказало такое большое влияние на сферу генеративного моделирования.

## Глубокие нейронные сети

Большинство систем глубокого обучения представляют собой *искусственные нейронные сети* (Artificial Neural Networks, ANN) или просто нейронные сети с несколькими скрытыми слоями, наложенными друг на друга. По этой причине термин *глубокое обучение* теперь стал почти синонимом термина *глубокие нейронные сети*. Отметим, любая система, использующая несколько слоев для выявления высокоуровневых представлений во входных данных, также является формой глубокого обучения (например, сети глубокого доверия (belief networks) и глубокие машины Больцмана (deep Boltzmann machines)).

Для начала рассмотрим в общих чертах, как глубокая нейронная сеть может генерировать прогнозы, опираясь на конкретные входные данные. Глубокая нейронная сеть состоит из последовательности слоев, наложенных друг на друга. Каждый слой содержит *узлы* (нейроны), связанные с узлами в предыдущем слое посредством набора *весов* (весовых коэффициентов). Как мы увидим далее, существует много разных типов слоев, но на практике чаще всего используются *полносвязанные* (dense) слои, соединяющие каждый свой узел со всеми узлами в предыдущем слое. При наложении слоев друг на друга узлы в каждом последующем слое получают возможность представлять все более сложные аспекты входных данных.

Например, слой 1 на рис. 2.2 состоит из узлов, активирующихся сильнее, когда обнаруживают определенные базовые признаки во входном изображении, такие как края и границы. Выходные данные из этих узлов затем

передаются в узлы уровня 2, которые могут использовать эту информацию для обнаружения чуть более сложных признаков, и т. д. Кульминацией этого процесса является выходной уровень в конце: он выводит набор чисел, которые можно преобразовать в вероятности принадлежности исходных данных к одной из *n* категорий. Волшебство глубоких нейронных сетей заключается в их способности находить наборы весов для каждого слоя, дающие наиболее точные прогнозы. Процесс поиска этих весов называется *обучением* сети.

В процессе обучения сети передаются пакеты изображений и полученные результаты сравниваются с истинными. Ошибка в прогнозировании затем распространяется обратно по сети и корректирует каждый набор весов на небольшую величину в направлении, улучшающем прогноз. Этот процесс так и называется: *обратное распространение*. Постепенно каждый узел становится квалифицированным в определении конкретного признака, что в конечном итоге помогает сети делать более точные прогнозы.

Глубокие нейронные сети могут иметь любое количество слоев в середине, которые называются *скрытыми*. Например, сеть ResNet[1], предназначенная для распознавания изображений, содержит 152 слоя. В главе 3 мы увидим,

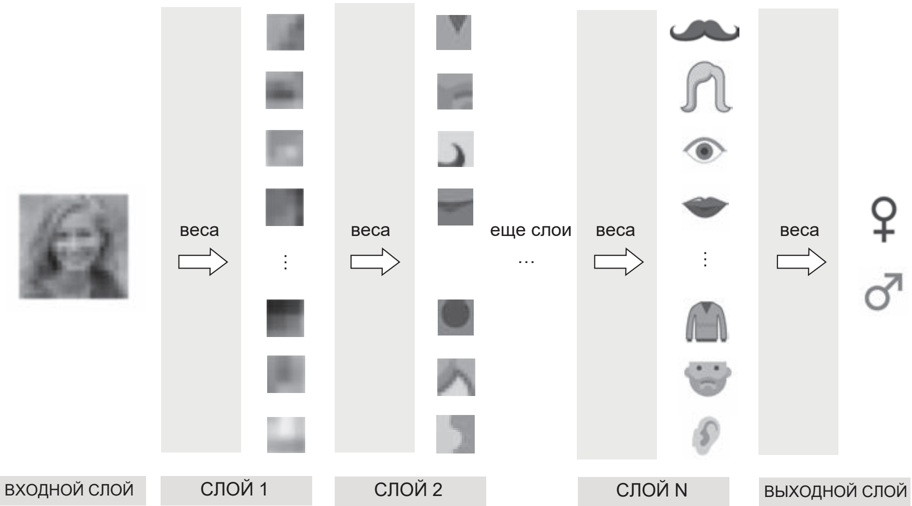



**Рис. 2.2.** Диаграмма, представляющая основную идею глубокого обучения

[1] Кайминг Хе (Kaiming He) и др., «Deep Residual Learning for Image Recognition», 10 декабря 2015, https://arxiv.org/abs/1512.03385.

что глубокие нейронные сети можно использовать для изменения высокоуровневых признаков изображений (цвет волос или выражение лица) путем ручной настройки этих скрытых слоев. Это возможно только потому, что наиболее глубокие слои сети выявляют высокоуровневые признаки, с которыми можно работать напрямую.

А теперь перейдем к практической стороне глубокого обучения и настроим Keras и TensorFlow — две библиотеки, которые позволят вам начать создавать свои собственные генеративные глубокие нейронные сети.

## Keras и TensorFlow

Keras — это высокоуровневая библиотека для Python, предназначенная для конструирования нейронных сетей, и основная библиотека, которую мы будем использовать в этой книге. Это чрезвычайно гибкая библиотека с очень удобным API, что делает ее идеальным выбором для начального знакомства с глубоким обучением. Кроме того, Keras предлагает множество строительных блоков, объединяя которые можно создавать очень сложные архитектуры глубокого обучения.

Keras не выполняет низкоуровневые операции с массивами, которые необходимы для обучения нейронных сетей. Для этой цели она использует одну из трех низкоуровневых библиотек: TensorFlow, CNTK и Theano. Вы можете выбрать ту, которая для вас удобнее или которая быстрее работает с сетями с конкретной архитектурой. В большинстве случаев не имеет большого значения, какой вариант вы выберете, потому что обычно вам не потребуется писать код, напрямую использующий низкоуровневые функции. В этой книге мы используем TensorFlow, так как она является наиболее широко распространенной и хорошо документированной.

TensorFlow — это библиотека для Python с открытым исходным кодом, предназначенная для машинного обучения и разработанная в Google. Сейчас это одна из библиотек, наиболее широко используемых в решениях машинного обучения, причем с особым упором на манипулирование тензорами (отсюда и такое название).

Тензоры в контексте глубокого обучения — это просто многомерные массивы, которые хранят данные, проходящие через сеть. Как мы увидим далее, понимание того, как каждый слой нейронной сети изменяет форму данных при их прохождении, является совершенно необходимым для истинного понимания механики глубокого обучения.

Если вы только начинаете знакомство с глубоким обучением, я настоятельно рекомендую выбрать Keras с низкоуровневой библиотекой TensorFlow как основные инструменты. Эти две библиотеки образуют мощную комбинацию, с помощью которой вы сможете построить любую сеть, какую только сможете представить, и предлагают простой в освоении API, что очень важно для быстрого воплощения новых идей и понятий.

## Ваша первая глубокая нейронная сеть

Для начала убедимся, насколько легко создаются глубокие нейронные сети с помощью Keras. В качестве рабочего примера используем блокнот 02_01_deep_learning_deep_neural_network.ipynb для Jupyter Notebook из репозитория с примерами для книги.

### Загрузка данных

В этом примере мы используем набор данных CIFAR-10, коллекцию из 60 000 цветных изображений 32 × 32 пиксела, входящий в состав дистрибутива библиотеки Keras. Каждое изображение принадлежит одному из десяти классов (рис. 2.3).

Следующий код загружает и масштабирует исходные данные:

```
import numpy as np
from keras.utils import to_categorical
from keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data() ❶

NUM_CLASSES = 10

x_train = x_train.astype('float32') / 255.0 ❷
x_test = x_test.astype('float32') / 255.0

y_train = to_categorical(y_train, NUM_CLASSES) ❸
y_test = to_categorical(y_test, NUM_CLASSES)
```

❶ Загрузка набора данных CIFAR-10. `x_train` и `x_test` — это массивы numpy с формой `[50000, 32, 32, 3]` и `[10000, 32, 32, 3]` соответственно. `y_train` и `y_test` — это массивы numpy с формой `[50000, 1]` и `[10000, 1]` соответственно, содержащие целочисленные метки от 0 до 9, которые определяют класс каждого изображения.

❷ Исходные данные представлены массивом пикселов, в котором каждый канал цвета каждого пиксела представлен целым числом от 0 до 255. Нейронные сети работают лучше, когда входные значения находятся в диапазоне от −1 до 1, поэтому мы должны разделить числа на 255.

❸ Также нужно изменить целочисленные метки изображений, превратив их в векторы прямого кодирования. Если целочисленная метка, обозначающая класс изображения, имеет значение $i$, то соответствующий ей вектор прямого кодирования имеет длину 10 (число классов), с нулями во всех элементах, кроме $i$-го, который равен 1. В связи с этим `y_train` и `y_test` получают новые формы: `[50000, 10]` и `[10000, 10]` соответственно.

**Рис. 2.3.** Примеры изображений из набора данных CIFAR-10[1]

[1] Источник: Алекс Крижевский (Alex Krizhevsky), «Learning Multiple Layers of Features from Tiny Images», 2009, https://www.cs.toronto.edu/~kriz/cifar.html.

Стоит особо отметить форму данных, представляющих изображения в `x_train`: `[50000, 32, 32, 3]`. Первое измерение этого массива определяет индекс изображения в наборе данных, второе и третье — размеры изображения[1], а последнее — индекс канала (то есть красный, зеленый или синий, потому что изображения хранятся в формате RGB). В этом наборе данных нет ни *столбцов*, ни *строк*, но есть *тензор* с четырьмя измерениями. Например, следующий элемент соответствует значению зеленого канала (1) пиксела в позиции (12,13) изображения с номером 54:

```
x_train[54, 12, 13, 1]
# 0.36862746
```

## Конструирование модели

Keras предлагает два способа определения структуры нейронной сети: в виде модели `Sequential` или с помощью функционального API.

Модель `Sequential` удобно использовать, когда нужно быстро определить линейную последовательность слоев (то есть когда один слой непосредственно следует за предыдущим без какого-либо ветвления). Однако многие модели, представленные в этой книге, требуют, чтобы слои передавали свои выходные данные сразу нескольким слоям, находящимся за ними, или, наоборот, чтобы слой получал входные данные от нескольких слоев перед ним.

Чтобы построить сеть с ветвлениями, нужно использовать намного более гибкий функциональный API. Я советую с самого начала использовать функциональный API библиотеки Keras, даже для создания линейных моделей, потому что это умение пригодится вам потом, когда ваши нейронные сети станут сложнее в архитектурном отношении. Функциональный API дает полную свободу в выборе архитектуры глубокой нейронной сети.

Для демонстрации различий между этими двумя подходами в листингах 2.1 и 2.2 показана одна и та же сеть, реализованная с использованием модели `Sequential` и функционального API. Опробуйте обе и обратите внимание на то, что они дают одинаковый результат.

**Листинг 2.1.** Реализация сети с использованием модели Sequential

```
from keras.models import Sequential
from keras.layers import Flatten, Dense
```

[1] Индексы строк и столбцов в пиксельной матрице изображения. — *Примеч. пер.*

```
model = Sequential([
    Dense(200, activation = 'relu', input_shape=(32, 32, 3)),
    Flatten(),
    Dense(150, activation = 'relu'),
    Dense(10, activation = 'softmax'),
])
```

**Листинг 2.2.** Реализация сети с использованием функционального API

```
from keras.layers import Input, Flatten, Dense
from keras.models import Model

input_layer = Input(shape=(32,32, 3))

x = Flatten()(input_layer)

x = Dense(units=200, activation = 'relu')(x)
x = Dense(units=150, activation = 'relu')(x)

output_layer = Dense(units=10, activation = 'softmax')(x)

model = Model(input_layer, output_layer)>
```

Здесь мы использовали три слоя разных типов: `Input`, `Flatten` и `Dense`.

Слой `Input` — это точка входа в сеть. С его помощью мы сообщаем сети, что каждый элемент входных данных имеет форму кортежа. Обратите внимание, мы не указываем размер пакета; в этом нет необходимости, потому что мы можем передать в слой `Input` любое количество изображений. Нам не нужно явно указывать размер пакета в определении входного слоя.

Затем мы «сплющиваем» входные данные, преобразуя их в вектор с помощью слоя `Flatten`. В результате получается вектор с длиной 3072 (= 32 × 32 × 3). Делается это по той причине, что последующий слой `Dense` требует, чтобы на его вход подавались данные в виде плоского вектора, а не в виде многомерного массива. Как мы увидим далее, другие типы слоев принимают многомерные массивы, поэтому, выбирая слой того или иного типа, вы должны знать форму его входных и выходных данных, чтобы понять, когда следует использовать `Flatten`.

Слой `Dense` является, пожалуй, преобладающим типом слоя в любой нейронной сети. Он содержит определенное количество узлов, полностью связанных с предыдущим слоем, то есть каждый узел в этом слое связан с каждым узлом в предыдущем слое через единственную связь, имеющую вес (который может быть положительным или отрицательным). На выходе каждый узел возвращает взвешенную сумму входных данных, полученных

от предыдущего слоя, которая затем передается следующему слою через нелинейную *функцию активации*. Функция активации играет важнейшую роль в способности нейронной сети обучаться – выявлять сложные признаки, а не просто выводить линейную комбинацию входных данных.

Существует много видов функций активации, но наиболее важными из них являются ReLU, sigmoid и softmax.

Функция активации *ReLU* (Rectified Linear Unit – блок линейной ректификации) возвращает ноль, если входное значение отрицательное, а в остальных случаях возвращает само входное значение. Функция активации *LeakyReLU* очень похожа на ReLU, но, в отличие от ReLU, для отрицательных входных значений она возвращает не ноль, а небольшое отрицательное число, пропорциональное входному значению. Узлы с функцией активации ReLU могут иногда *умирать*, если всегда возвращают ноль из-за большого смещения преактивации в сторону отрицательных значений. В этом случае градиент получается равным нулю и через этот узел не происходит обратного распространения ошибки. Функция активации LeakyReLU решает проблему, гарантируя, что градиент не будет равен нулю. В настоящее время считается, что функции, основанные на ReLU, являются наиболее надежными инструментами для передачи информации между слоями глубокой сети и стабильности обучения.

Функция активации *sigmoid* может пригодиться, когда требуется, чтобы выходные значения масштабировались в диапазоне между 0 и 1, как, например, в задачах бинарной классификации с одним выходным узлом или в задачах классификации с несколькими метками, когда каждое наблюдение может одновременно принадлежать нескольким классам. Для сравнения на рис. 2.4 показаны графики функций активации ReLU, LeakyReLU и sigmoid.

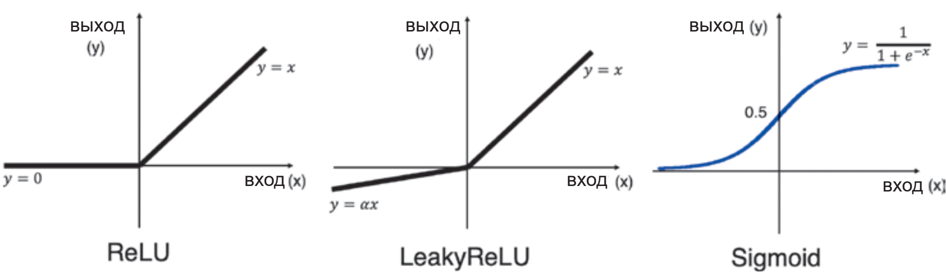



**Рис. 2.4.** Функции активации ReLU, LeakyReLU и sigmoid

Функция активации *softmax* может пригодиться, когда требуется, чтобы общая сумма выходных значений слоя была равна единице, как, например, в задачах многоклассовой классификации, когда каждое наблюдение может принадлежать только к одному классу. Она определяется как

$$y_i = \frac{e^{x_i}}{\Sigma_{j=1}^{J} e^{x_j}}.$$

Здесь $J$ — общее число узлов в слое. В нашей нейронной сети мы используем активацию softmax в последнем слое для получения на выходе набора из десяти вероятностей, сумма которых равна единице. Эти вероятности можно интерпретировать как вероятности принадлежности изображения каждому классу. В Keras функции активации можно также определять в отдельном слое, например:

```
x = Dense(units=200)(x)
x = Activation('relu')(x)
```

Этот код эквивалентен следующему:

```
x = Dense(units=200, activation = 'relu')(x)
```

В нашем примере мы пропускаем входные данные через два полносвязанных скрытых слоя, первый — с 200 узлами, а второй — с 150, и оба с функциями активации ReLU (рис. 2.5).

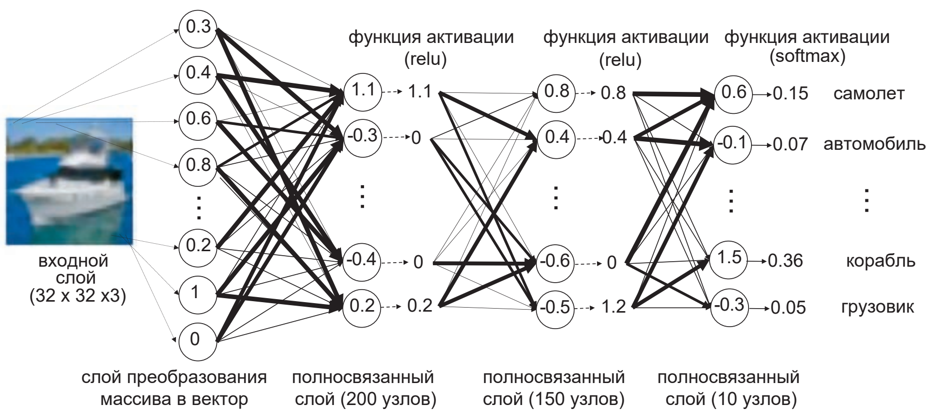



**Рис. 2.5.** Схема нейронной сети, обучающейся на наборе данных CIFAR-10

Последний шаг — определение самой модели с использованием класса Model. В Keras модель определяется входным и выходным слоями. В данном случае имеется один входной слой, который мы определили выше, а выходной слой — это последний слой Dense с десятью узлами. Также есть возможность определить модель с несколькими входными и выходными слоями; как это делается, мы увидим далее.

Форма входного слоя в нашем примере соответствует форме x_train, а форма выходного слоя Dense соответствует форме y_train. Убедиться в этом можно с помощью метода model.summary(), который выводит формы всех слоев в сети, как показано на рис. 2.6.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               614600
_________________________________________________________________
dense_2 (Dense)              (None, 150)               30150
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1510
=================================================================
Total params: 646,260
Trainable params: 646,260
Non-trainable params: 0
_________________________________________________________________
```

**Рис. 2.6.** Сводная информация о сети

Обратите внимание: чтобы показать, что количество наблюдений, которые будут переданы в сеть, пока неизвестно, Keras использует маркер None. На самом деле это число и не нужно; мы можем передать в сеть и одно наблюдение, и тысячу, потому что тензорные операции проводятся по всем наблюдениям одновременно с использованием алгоритмов линейной алгебры, которые реализует библиотека TensorFlow. Этим же обусловлено и увеличение производительности, когда обучение глубоких нейронных сетей производится на графических процессорах вместо обычных: графические процессоры оптимизированы для умножения больших тензоров, потому что эти вычисления также необходимы для сложных манипуляций с графикой.

Метод `summary` также выводит количество параметров (весов), которые будут обучаться в каждом слое. Если вы обнаружите, что ваша модель обучается слишком медленно, то выведите сводную информацию, чтобы увидеть, есть ли в сети какие-нибудь слои с огромным количеством весов. Если такие слои имеются, то подумайте над тем, можно ли уменьшить число узлов в слое, чтобы ускорить обучение.

## Компиляция модели

На этом этапе производится компиляция модели с заданными оптимизатором и функцией потерь:

```
from keras.optimizers import Adam

opt = Adam(lr=0.0005)
model.compile(loss='categorical_crossentropy', optimizer=opt,
              metrics=['accuracy'])
```

Функция потерь используется нейронной сетью для сравнения ее прогноза с истиной. Она возвращает одно число для каждого наблюдения; чем больше это число, тем хуже сеть справилась с прогнозированием этого наблюдения.

Keras предлагает множество встроенных функций потерь, а также дает вам возможность создавать свои собственные. На практике чаще всего используются три функции: среднеквадратичная ошибка, многозначная перекрестная энтропия и бинарная перекрестная энтропия. Важно понимать, когда лучше использовать каждую из них.

Если ваша нейронная сеть предназначена для решения задачи регрессии (то есть выходные данные представляют значения из непрерывной области), то можно использовать функцию потери — *среднеквадратичную ошибку*. Это среднее значение суммы квадратов разностей между истинным значением $y_i$ и прогнозируемым $p_i$ в каждом выходном узле, где среднее берется по всем $n$ выходным узлам:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - p_i\right)^2.$$

Если нейронная сеть предназначена для решения задачи классификации, где каждое наблюдение может относиться только к одному классу, то правильнее будет выбрать функцию потерь, известную как *многозначная перекрестная энтропия*. Она определяется следующим образом:

$$-\sum_{i=1}^{n} y_i \log\left(p_i\right).$$

Наконец, если нейронная сеть предназначена для решения задачи бинарной классификации с одним выходным узлом или классификации с несколькими метками, когда каждое наблюдение может одновременно принадлежать нескольким классам, то следует использовать бинарную перекрестную энтропию:

$$-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log\left(p_i\right) + \left(1 - y_i\right)\log\left(1 - p_i\right)\right).$$

Оптимизатор — это алгоритм, используемый для обновления весов в нейронной сети на основе градиента функции потерь. Одним из наиболее часто используемых и стабильных оптимизаторов является *Adam*[1]. В большинстве случаев вам не придется настраивать параметры по умолчанию оптимизатора Adam, за исключением скорости обучения. Чем выше скорость обучения, тем сильнее будут меняться веса на каждом шаге обучения. Обучение при более высокой скорости изначально происходит быстрее, но оно протекает менее стабильно и может не найти минимум функции потери. Этот параметр можно настраивать и корректировать во время обучения.

Другой распространенный оптимизатор, с которым вы можете столкнуться, — *RMSProp*. Работая с ним, вам также не придется заниматься кропотливой настройкой его параметров, но я советую прочитать документацию Keras (https://keras.io/optimizers), чтобы понять роль каждого параметра.

В вызов метода модели `compile` передается функция потерь, оптимизатор и параметр `metrics`, в котором можно передать любые дополнительные метрики, которые хотелось бы передать в процесс обучения, например точность.

[1] Дидерик Кингма (Diederik Kingma) и Джимми Ба (Jimmy Ba), «Adam: A Method for Stochastic Optimization», 22 декабря 2014, https://arxiv.org/abs/1412.6980v8.

## Обучение модели

До сих пор мы не передали в модель никаких данных, а только настроили архитектуру и скомпилировали модель с функцией потерь и оптимизатором. Чтобы обучить модель, достаточно вызвать метод `fit`:

```
model.fit(x_train ❶
          , y_train ❷
          , batch_size = 32 ❸
          , epochs = 10 ❹
          , shuffle = True ❺
          )
```

❶ Исходные данные — массив с изображениями.

❷ Метки классов в формате прямого кодирования.

❸ Параметр `batch_size` определяет, сколько наблюдений будет передаваться в сеть на каждом шаге обучения.

❹ Параметр `epochs` определяет, сколько раз сеть будет просматривать полный комплект обучающих данных.

❺ Если `shuffle = True`, то пакеты обучающих данных будут перемешиваться случайным образом перед каждым шагом обучения.

Код запускает обучение глубокой нейронной сети предсказанию категории изображения из набора данных CIFAR-10. В процессе обучения веса сети сначала инициализируются небольшими случайными значениями. Затем сеть выполняет серию шагов обучения.

На каждом шаге обучения в сеть передается один пакет изображений и в ходе обратного распространения ошибки корректируются веса. Параметр `batch_size` определяет количество изображений в каждом пакете. Чем больше размер пакета, тем стабильнее результат вычисления градиента, но каждый шаг обучения выполняется медленнее. Если для вычисления градиента на каждом шаге обучения использовать весь набор данных, то потребуется слишком много времени и вычислительных ресурсов, поэтому размер пакета обычно выбирается в диапазоне от 32 до 256. Также рекомендуется постепенно увеличивать размер пакета в процессе обучения.[1]

[1] Сэмюэль Л. Смит (Samuel L. Smith) и др., «Don’t Decay the Learning Rate, Increase the Batch Size», 1 ноября 2017, https://arxiv.org/abs/1711.00489.

```
model.fit(x_train
          , y_train
          , batch_size=BATCH_SIZE
          , epochs=EPOCHS
          , shuffle=True)
```

```
Epoch 1/10
50000/50000 [==============================] - 8s 164us/step - loss: 1.8424 - acc: 0.3354
Epoch 2/10
50000/50000 [==============================] - 8s 154us/step - loss: 1.6592 - acc: 0.4048
Epoch 3/10
50000/50000 [==============================] - 8s 153us/step - loss: 1.5733 - acc: 0.4381
Epoch 4/10
50000/50000 [==============================] - 8s 154us/step - loss: 1.5232 - acc: 0.4579
Epoch 5/10
50000/50000 [==============================] - 8s 155us/step - loss: 1.4874 - acc: 0.4698
Epoch 6/10
50000/50000 [==============================] - 8s 165us/step - loss: 1.4569 - acc: 0.4799
Epoch 7/10
50000/50000 [==============================] - 10s 208us/step - loss: 1.4281 - acc: 0.4887
Epoch 8/10
50000/50000 [==============================] - 8s 165us/step - loss: 1.4038 - acc: 0.4984
Epoch 9/10
50000/50000 [==============================] - 8s 153us/step - loss: 1.3797 - acc: 0.5084
Epoch 10/10
50000/50000 [==============================] - 8s 155us/step - loss: 1.3571 - acc: 0.5187
```

**Рис. 2.7.** Вывод метода fit

Так продолжается, пока сеть не увидит все наблюдения, имеющиеся в наборе данных. Этим завершается первая *эпоха*. Затем начинается вторая эпоха, и данные вновь передаются в сеть пакетами. Этот процесс повторяется до тех пор, пока не будет достигнуто заданное число эпох.

В процессе обучения Keras информирует о ходе выполнения, как показано на рис. 2.7. Как видите, в этом примере обучающий набор данных из 50 000 наблюдений был обработан сетью 10 раз (то есть выполнено 10 эпох) со скоростью примерно 160 микросекунд на наблюдение. Потеря многозначной перекрестной энтропии снизилась с 1,842 до 1,357, что привело к увеличению точности с 33,5 % после первой эпохи до 51,9 % после десятой.

## Оценка модели

Мы знаем, что модель достигла точности 51,9 % на обучающем наборе, но насколько хорошо она справится с данными, которые никогда не видела прежде? Чтобы ответить на этот вопрос, можно использовать метод `evaluate` из библиотеки Keras:

```
model.evaluate(x_test, y_test)
```

На рис. 2.8 показано, что вывел этот метод.

```
10000/10000 [==============================] - 1s 55us/step

[1.4358007415771485, 0.4896]
```

**Рис. 2.8.** Вывод метода evaluate

Этот метод выводит список отслеживаемых метрик: многозначная перекрестная энтропия и точность. Точность модели составляет 49 % на изображениях, которые она никогда раньше не видела. Обратите внимание: если бы модель угадывала случайным образом, то она имела бы точность около 10 % (потому что имеется 10 классов), поэтому 50 % — хороший результат, учитывая, что мы использовали очень простую нейронную сеть. Мы можем просмотреть некоторые прогнозы для изображений в контрольном наборе, использовав метод `predict`:

```
CLASSES = np.array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
                    'frog', 'horse', 'ship', 'truck'])

preds = model.predict(x_test) ❶
preds_single = CLASSES[np.argmax(preds, axis = -1)] ❷
actual_single = CLASSES[np.argmax(y_test, axis = -1)]
```

❶ `preds` — это массив с формой `[10000, 10]`, то есть вектор с десятью вероятностями принадлежности к разным классам для каждого наблюдения.

❷ Этот массив вероятностей преобразуется в единственное предсказание с использованием функции `argmax` из `numpy`. Здесь `axis = -1` подразумевает функцию свертки массива по последнему измерению (измерению классов), то есть `preds_single` имеет форму `[10000, 1]`.

Мы можем просмотреть некоторые изображения вместе с их метками и прогнозами, выполнив следующий код. Как и ожидалось, точность прогнозирования составляет примерно 50 %:

```
import matplotlib.pyplot as plt

n_to_show = 10
indices = np.random.choice(range(len(x_test)), n_to_show)

fig = plt.figure(figsize=(15, 3))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
```

```
for i, idx in enumerate(indices):
    img = x_test[idx]
    ax = fig.add_subplot(1, n_to_show, i+1)
    ax.axis('off')
    ax.text(0.5, -0.35, 'pred = ' + str(preds_single[idx]), fontsize=10
       , ha='center', transform=ax.transAxes)
    ax.text(0.5, -0.7, 'act = ' + str(actual_single[idx]), fontsize=10
       , ha='center', transform=ax.transAxes)
    ax.imshow(img)
```

На рис. 2.9 показаны некоторые случайно выбранные изображения с прогнозами модели и истинными метками.

Поздравляю! Вы только что сконструировали свою первую глубокую нейронную сеть с помощью Keras и использовали ее для прогнозирования новых данных. Несмотря на то что эта задача является задачей обучения с учителем, в последующих главах (при рассмотрении вопросов создания генеративных моделей) многие из основных понятий (функции потерь, функции активации и формы слоев и др.) не утратят своей важности. Теперь поссмотрим на возможные пути улучшения этой модели введением нескольких слоев нового типа.

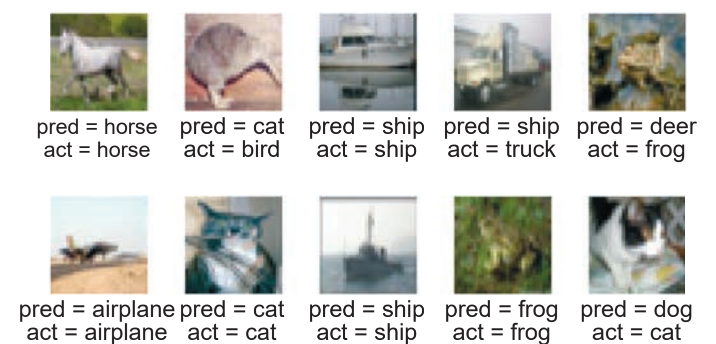



**Рис. 2.9.** Некоторые прогнозы, сделанные моделью, и фактические метки

## Улучшение модели

Одна из причин того, почему наша сеть работает не так хорошо, как могла бы, заключается в том, что в ней нет ничего, что учитывало бы пространственную структуру входных изображений. Фактически на первом шаге мы «сплющиваем» изображения в вектор, чтобы передать их первому слою `Dense`! Решить проблему можно, используя *сверточный слой*.

## Сверточные слои

Что подразумевается под *сверткой* в контексте глубокого обучения? На рис. 2.10 показано, как выполняется свертка фрагмента 3 × 3 × 1 черно-белого изображения с использованием *фильтра* (или *ядра*) 3 × 3 × 1.

Свертка осуществляется путем попиксельного умножения фильтра на фрагмент изображения и последующего суммирования результатов. Чем ближе фрагмент изображения соответствует фильтру, тем больше результат, и наоборот, чем меньше фрагмент соответствует фильтру, тем меньше результат. Перемещая фильтр по всему изображению, слева направо и сверху вниз, и записывая результаты свертки, мы получим новый массив, в котором резко будут выделяться конкретные особенности входного изображения в зависимости от значений в фильтре.

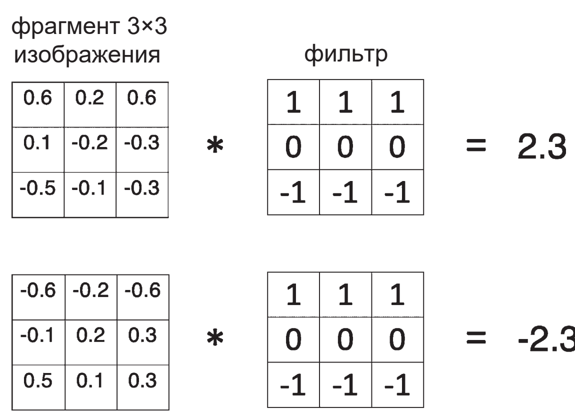



**Рис. 2.10.** Операция свертки

Именно для этого и предназначен сверточный слой, но он содержит несколько фильтров, а не один. Например, на рис. 2.11 показаны два фильтра, выделяющие горизонтальные и вертикальные границы. Увидеть, как протекает этот сверточный процесс, можно в блокноте 02_02_deep_learning_convolutions.ipynb из репозитория с примерами для книги. При работе с цветными изображениями каждый фильтр должен иметь три канала, а не

один (то есть каждый фильтр должен иметь форму 3 × 3 × 3), что соответствует трем каналам (красный, зеленый, синий) изображения. Слой `Conv2D` в Keras применяет свертки к входному тензору с двумя пространственными измерениями (изображение как раз является таким тензором). Например, вот как выглядит код, использующий Keras, который соответствует диаграмме на рис. 2.11:

```
input_layer = Input(shape=(64,64,1))

conv_layer_1 = Conv2D(
    filters = 2
    , kernel_size = (3,3)
    , strides = 1
    , padding = «same»
    )(input_layer)
```

| 1 | 1 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| –1 | –1 | –1 |

входной слой
1 x 64 x 64 x 1
размер пакета × высота × ширина × число фильтров

| –1 | 0 | 1 |
|---|---|---|
| –1 | 0 | 1 |
| –1 | 0 | 1 |

2 фильтра (каждый 3 × 3 × 1)

результат
1 x 64 x 64 x 2
размер пакета × высота × ширина × число фильтров

**Рис. 2.11.** Два сверточных фильтра применяются к черно-белому изображению

**ПАРАМЕТР STRIDES**

Параметр `strides` определяет величину шага перемещения фильтров по входным данным. Чем больше шаг, тем меньше размер выходного тензора. Например, при `strides = 2` высота и ширина выходного тензора будут вдвое меньше высоты и ширины входного. Это важно для уменьшения пространственных размеров тензора при прохождении через сеть при одновременном увеличении количества каналов.

**ПАРАМЕТР PADDING**

Если задан параметр `padding = "same"`, то входные данные будут дополняться нулями, чтобы размер выходного тензора в точности совпадал с размером входного при `strides = 1`. На рис. 2.12 показано ядро 3 × 3, которое перемещается по входному изображению 5 × 5, с параметрами `padding = "same"` и `strides = 1`. На выходе этого сверточного слоя получится тензор с тем же размером 5 × 5, так как дополнение позволяет ядру расширить границы изображения на краях, чтобы можно было применить его пять раз в обоих направлениях. Без дополнения ядро было бы применено только три раза вдоль каждого направления, и в результате получился бы тензор 3 × 3.

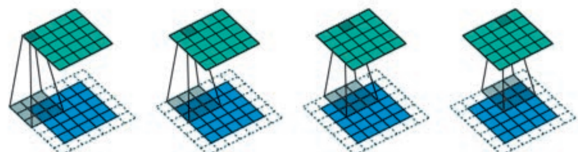


**Рис. 2.12.** Ядро 3 × 3 × 1 (серое) перемещается по входному изображению 5 × 5 × 1 (синее) с параметрами padding="same" и strides = 1, в результате получается выходной тензор 5 × 5 × 1 (зеленый)[1]

Параметр `padding = "same"` позволяет контролировать размер тензора по мере прохождения через множество сверточных слоев.

[1] Источник: Винсент Дюмулен (Vincent Dumoulin) и Франческо Висин (Francesco Visin), «A Guide to Convolution Arithmetic for Deep Learning», 12 января 2018, https://arxiv.org/pdf/1603.07285.pdf.

Значения в фильтрах — это веса, которые определяются нейронной сетью в процессе обучения. Первоначально они выбираются случайным образом, но постепенно адаптируются, обретая интересные особенности, такие как границы или определенные цветовые комбинации.

Результатом слоя `Conv2D` является другой четырехмерный тензор, имеющий форму (размер пакета, высота, ширина, число фильтров), поэтому слои `Conv2D` можно помещать друг за другом и тем самым увеличивать глубину нейронной сети. Важно понимать, как меняется форма тензора при переходе данных от одного сверточного слоя к следующему. Для наглядности представим, что мы применяем слои `Conv2D` к набору данных CIFAR-10. На этот раз вместо одного канала (определяющего оттенок серого) у нас их будет три (красный, зеленый и синий).

На рис. 2.13 показана сеть, которая реализуется следующим кодом:

```
input_layer = Input(shape=(32,32,3))

conv_layer_1 = Conv2D(
    filters = 10
    , kernel_size = (4,4)
    , strides = 2
    , padding = 'same'
    )(input_layer)

conv_layer_2 = Conv2D(
    filters = 20
    , kernel_size = (3,3)
    , strides = 2
    , padding = 'same'
    )(conv_layer_1)

flatten_layer = Flatten()(conv_layer_2)

output_layer = Dense(units=10, activation = 'softmax')(flatten_layer)

model = Model(input_layer, output_layer)
```

Увидеть, как меняется форма тензора при прохождении через сеть, можно с помощью метода `model.summary()`(рис. 2.14).

Проанализируем эту сеть от входа до выхода. Входные данные имеют форму `(None, 32, 32, 3)` — Keras использует `None`, чтобы показать, что мы можем передать в сеть сразу любое количество изображений. Поскольку сеть просто выполняет операции тензорной алгебры, мы можем передавать изображения в сеть *пакетами*, а не по отдельности. В первом сверточном

слое фильтры имеют форму 4 × 4 × 3, так как мы выбрали фильтр с высотой и шириной 4 (`kernel_size = (4,4)`) и в предыдущем слое данные имеют три канала (красный, зеленый и синий). Следовательно, число параметров (или весов) в слое составляет $(4 \times 4 \times 3 + 1) \times 10 = 490$, где слагаемое + 1 обусловлено включением члена смещения, присоединенного к каждому из фильтров. Отметим, что глубина фильтров в слое всегда равна количеству каналов в предыдущем слое.

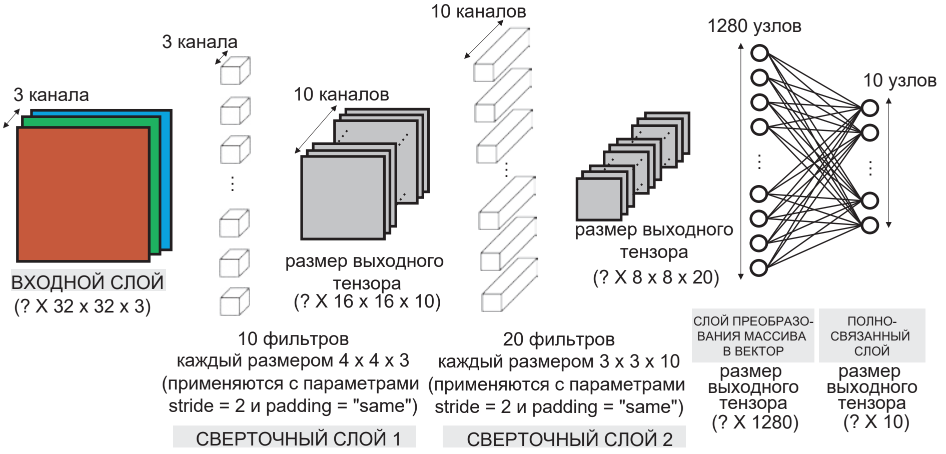



**Рис. 2.13.** Схема сверточной нейронной сети

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 10)        490
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 20)          1820
_________________________________________________________________
flatten_1 (Flatten)          (None, 1280)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                12810
=================================================================
Total params: 15,120
Trainable params: 15,120
Non-trainable params: 0
_________________________________________________________________
```

**Рис. 2.14.** Сводная информация о сверточной нейронной сети

Как и прежде, результатом применения каждого фильтра к каждому фрагменту $4 \times 4 \times 3$ входного изображения является сумма попиксельных произведений весов фильтра на значения каналов во фрагменте изображения, который покрывает этот фильтр. Так как заданы параметры `strides = 2` и `padding = "same"`, ширина и высота выходного тензора в два раза меньше ширины и высоты входного тензора и равны 16, а поскольку имеется десять фильтров, выход первого слоя представляет собой пакет тензоров, каждый из которых имеет форму `[16, 16, 10]`. В общем случае выходной тензор сверточного слоя с параметром `padding="same"` имеет форму:

$$\left(None, \frac{\textit{высота входного изображения}}{\textit{размер шага}}, \frac{\textit{ширина входного изображения}}{\textit{размер шага}}, \textit{число фильтров}\right).$$

Во втором сверточном слое мы выбрали форму $3 \times 3$ для фильтров, и теперь они имеют глубину 10, согласно количеству каналов в предыдущем слое. Поскольку в этом слое у нас 20 фильтров, общее количество параметров (весов) получается равным $(3 \times 3 \times 10 + 1) \times 20 = 1820$. И снова мы используем параметры слоя `strides = 2` и `padding = "same"`, из-за чего ширина и высота выходного тензора уменьшаются вдвое. Таким образом, выходной тензор имеет форму `(None, 8, 8, 20)`. После применения последовательности слоев `Conv2D` нужно преобразовать многомерный тензор в вектор, использовав слой `Flatten`. В результате получается набор из $8 \times 8 \times 20 = 1280$ элементов, который можно передать в заключительный слой `Dense` с десятью узлами и функцией активации softmax, представляющими вероятность принадлежности изображения к каждой категории в задаче классификации с десятью категориями.

Этот пример наглядно демонстрирует, как из нескольких сверточных слоев создать сверточную нейронную сеть. Но прежде чем показать, насколько эта сеть эффективнее распознает изображения по сравнению с полносвязанной нейронной сетью, взглянем еще на два типа слоев, также позволяющих увеличить качество прогнозирования: `BatchNormalization` и `Dropout`.

## Пакетная нормализация

Одна из проблем, часто возникающих при обучении глубоких нейронных сетей, заключается в сохранении значений весов в разумных пределах — если они становятся слишком большими, это признак того, что сеть страдает от

так называемого *взрыва градиента*. Поскольку ошибки распространяются через сеть в обратном направлении, градиент, вычисляемый в ранних слоях, может иногда расти в экспоненциальной прогрессии, вызывая гигантские колебания значений веса. Если ваша функция потерь начинает возвращать `NaN` (Not a Number — не число), то, скорее всего, ваши веса выросли настолько, что стали вызывать ошибку переполнения.

Подобное может происходить не сразу после начала обучения сети. Иногда сеть может благополучно обучаться часами, и вдруг функция потерь внезапно возвращает `NaN`, и сеть взрывается. Это особенно неприятно, когда сеть, казалось бы, успешно обучается в течение длительного времени. Чтобы этого не случилось, нужно понять причину взрыва градиента.

Одной из таких причин является обеспечение стабильного начала обучения в течение первых нескольких итераций. Поскольку веса сети изначально выбираются случайно, немасштабированные входные данные потенциально могут привести к получению огромных значений активации, которые немедленно приводят к взрыву градиента. Например, вместо передачи фактических значений пикселов, изменяющихся в диапазоне от 0 до 255, мы обычно масштабируем эти значения и приводим к диапазону от –1 до 1.

Поскольку входные данные масштабируются, естественно ожидать, что активации во всех последующих слоях тоже будут достаточно хорошо масштабироваться. В первых итерациях это действительно может быть так, но по мере обучения сети веса будут отодвигаться все дальше от своих случайных начальных значений, и в результате предположение может перестать соответствовать действительности. Это явление известно как *сдвиг переменных* (covariate shift).

Представьте, что вы несете высокую стопку книг, и вдруг налетает ветер. Вы наклоняете стопку навстречу ветру, чтобы компенсировать его воздействие, но при этом некоторые книги сдвигаются, так что вся стопка становится менее устойчивой. Первое время вы продолжаете нести стопку, не испытывая особых проблем, но с каждым порывом стопка становится все более и более неустойчивой и наконец падает. Это и есть сдвиг переменных. Вернемся к нейронным сетям: каждый их слой подобен книге в стопке. Чтобы не потерять стабильность, когда сеть обновляет веса, каждый слой неявно предполагает, что распределение входных данных, получаемых от предыдущего слоя, приблизительно одинаково во всех итерациях. Однако

поскольку ничто не может предотвратить значительное смещение распределения какой-либо из активаций в определенном направлении, иногда это может приводить к неконтролируемым изменениям значений весов и общему коллапсу сети.

*Пакетная нормализация* значительно смягчает эту проблему. Решение выглядит на удивление простым. Слой пакетной нормализации вычисляет среднее значение и стандартное отклонение для каждого из входных каналов в пакете и нормализует эти показатели, вычитая среднее значение и деля его на стандартное отклонение. Для каждого канала есть два обучаемых параметра: масштаб (гамма) и сдвиг (бета). Выход слоя — это всего лишь нормализованный вход, масштабированный по величине гаммы и сдвинутый на величину бета (рис. 2.15).

Слой пакетной нормализации можно поместить после полносвязанных или сверточных слоев, чтобы нормализовать их выход. Это похоже на обвязку книг в стопке, гарантирующую, что с течением времени в их положении не произойдут огромные сдвиги.

**Вход:** значения x в пакете $\mathcal{B} = \{x_{1...m}\}$;
Параметры для обучения: $\gamma, \beta$
**Выход:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$ // среднее по пакету

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$ // стандартное отклонение по пакету

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$ // нормализация

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$ // масштабирование и сдвиг

**Алгоритм 1:** Batch Normalizing Transform (Преобразование пакетной нормализации), применяется к активации x в мини-пакете.

**Рис. 2.15.** Процесс пакетной нормализации[1]

[1] Источник: Сергей Йоффе (Sergey Ioffe) и Кристиан Жегеди (Christian Szegedy), «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», 11 февраля 2015, https://arxiv.org/abs/1502.03167.

Как этот слой работает во время тестирования? На этапе прогнозирования может понадобиться спрогнозировать только одно наблюдение, то есть не существует пакета, для которого можно вычислить средние значения. Чтобы обойти эту проблему, во время обучения слой пакетной нормализации также вычисляет скользящее среднее от среднего значения и стандартного отклонения каждого канала и сохраняет его в слое для использования во время тестирования.

Сколько параметров содержится в слое пакетной нормализации? Для каждого канала в предыдущем слое определяются два веса: масштаб (гамма) и сдвиг (бета). Это — *обучаемые* параметры. Скользящее среднее и стандартное отклонение вычисляются для каждого канала на основе данных, проходящих через слой, а не на основе ошибок, распространяющихся в обратном направлении, и потому называются *необучаемыми* параметрами. Таким образом, из четырех параметров для каждого канала в предыдущем слое два являются обучаемыми, а два — необучаемыми.

В Keras пакетную нормализацию реализует слой `BatchNormalization`:

```
BatchNormalization(momentum = 0.9)
```

Параметр `momentum` — вес, придаваемый предыдущему значению при вычислении скользящего среднего и стандартного отклонения.

## Слои прореживания

При подготовке к экзамену, чтобы подкрепить знания, студенты обычно используют свои конспекты и примерные вопросы. Некоторые пытаются запомнить ответы на эти вопросы, но затем испытывают трудности на экзамене, потому что не понимают сути вопроса. Более успешные студенты читают учебники и решают практические задачи, чтобы углубить свое понимание предмета и суметь правильно ответить на вопросы, которые прежде им не задавались.

Тот же принцип действует и в машинном обучении. Любой успешный алгоритм машинного обучения должен уметь обобщать полученные знания, прежде невиданные им данные, а не просто *запоминать* набор обучающих данных. Если алгоритм показывает хорошие результаты на наборе обучающих данных, но часто ошибается на контрольном наборе данных, то очевидно, что он страдает от *переобучения*. Для смягчения этой проблемы

используются методы *регуляризации*, гарантирующие, что модель будет оштрафована, если начнет переобучаться.

Есть много способов регуляризации алгоритма машинного обучения, но в глубоком обучении чаще всего используется прием, основанный на применении слоев *прореживания*. Эта идея была представлена Джеффри Хинтоном (Geoffrey Hinton) в 2012 году, а двумя годами позже описана в статье Шриваставы (Srivastava) и его соавторов.[1] Слои прореживания устроены очень просто. Во время обучения каждый прореживающий слой выбирает случайный набор узлов из предыдущего слоя и устанавливает их выходные значения равными нулю (рис. 2.16).

Эта простая манипуляция значительно снижает вероятность переобучения, гарантируя, что сеть не станет чрезмерно зависимой от определенных узлов или их групп, которые, по сути, просто запоминают наблюдения из обучающего набора. При добавлении прореживающих слоев сеть больше не может слишком полагаться на какой-то один узел, поэтому знания распределяются по всей сети более равномерно. Так модель обретает способность лучше обобщать прежде не встречавшиеся ей данные, поскольку была обучена давать точные прогнозы даже в незнакомых условиях, вызванных сбросом значений случайных узлов. В прореживающем слое нет весов для обучения — отбрасываемые узлы определяются стохастически (случайно). На этапе тестирования прореживающий слой ничего не отбрасывает, поэтому для прогнозирования используется вся сеть.

Это напоминает то, как студент-математик читает конспекты, случайно выбирая ключевые формулы, отсутствующие в справочнике (учебнике). Так студенты учатся отвечать на вопросы, применяя основные принципы, а не отыскивая готовые решения. На экзамене им будет гораздо проще отвечать на вопросы, которые прежде им не задавались, опираясь на свои способности проводить обобщения. Слой `Dropout` в Keras реализует эту функцию с помощью параметра `rate`, определяющего долю отбрасываемых узлов из предыдущего слоя:

```
Dropout(rate = 0.25)
```

[1] Нитиш Шривастава (Nitish Srivastava) и др., «Dropout: A Simple Way to Prevent Neural Networks from Overfitting», Journal of Machine Learning Research 15 (2014): 1929–1958, http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

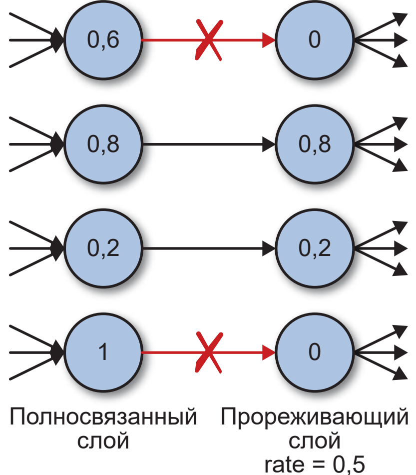



**Рис. 2.16.** Прореживающий слой

Обычно слои `Dropout` используются после слоев `Dense` как наиболее склонные к переобучению из-за большого количества весов. Но их можно использовать и после сверточных слоев.

Пакетная нормализация, представленная выше, тоже уменьшает вероятность переобучения, и поэтому многие современные архитектуры глубокого обучения вообще не используют прореживание, применяя для регуляризации только пакетную нормализацию. Как и в случае с большинством принципов глубокого обучения, не существует золотого правила, применимого в любой ситуации. Единственный способ точно узнать, что лучше, — протестировать разные архитектуры и определить, какие лучше всего работают с контрольным набором данных.

## Соединяем все вместе

Итак, вы познакомились с тремя новыми типами слоев в Keras: `Conv2D`, `BatchNormalization` и `Dropout`. Давайте объединим их в новую архитектуру глубокого обучения и посмотрим, насколько хорошо она справится с задачей классификации набора данных CIFAR-10. Для этого можно воспользоваться блокнотом 02_03_deep_learning_conv_neural_network.ipynb для Jupyter Notebook

из репозитория с примерами для книги. Вот как выглядит архитектура модели, которую мы проверим:

```
input_layer = Input((32,32,3))

x = Conv2D(filters = 32, kernel_size = 3
        , strides = 1, padding = 'same')(input_layer)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 32, kernel_size = 3, strides = 2, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 2, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Flatten()(x)

x = Dense(128)(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = Dropout(rate = 0.5)(x)

x = Dense(NUM_CLASSES)(x)
output_layer = Activation('softmax')(x)

model = Model(input_layer, output_layer)
```

Здесь используются четыре слоя `Conv2D`, за каждым из которых следуют слои `BatchNormalization` и `LeakyReLU`. После преобразования полученного тензора в вектор мы пропускаем данные через слой `Dense` с размером 128, за которым снова следуют `BatchNormalization` и `LeakyReLU`. Далее следует слой `Dropout` для регуляризации, а завершается сеть выходным слоем `Dense` с размером 10.

Порядок следования слоев `BatchNormalization` и `Activation` — это во многом вопрос личных предпочтений. Мне нравится помещать `BatchNormalization` перед `Activation`, но в некоторых успешных архитектурах эти слои применяются в обратном порядке. Если вы решите использовать `BatchNormalization` перед `Activation`, то запомнить порядок вам поможет мнемоническое сокращение *BAD* (`BatchNormalization`, `Activation` затем `Dropout`)!

На рис. 2.17 показана сводная информация о модели.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 32)        9248
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 64)          256
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 8, 8, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               524416
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 592,554
Trainable params: 591,914
Non-trainable params: 640
_________________________________________________________________
```

**Рис. 2.17.** Сверточная нейронная сеть (Convolutional Neural Network, CNN) для классификации CIFAR-10

Прежде чем двигаться дальше, проверьте, сможете ли вы вручную вычислить форму выходного тензора и количество параметров для каждого слоя. Это хороший способ доказать себе, что вы полностью поняли, как устроен каждый слой и как он связан с предыдущим слоем! Не забудьте про веса смещений, которые включены в состав слоев `Conv2D` и `Dense`.

Скомпилируем и обучим модель так же, как и раньше, и вызовем метод `evaluate`, чтобы определить ее точность на контрольном наборе (рис. 2.18).

Как видите, этой модели удалось достичь точности 71,5 %, что можно считать большим успехом по сравнению с точностью 49,0 %, отмеченной у предыдущей модели (рис. 2.19).

```
model.evaluate(x_test, y_test, batch_size=1000)

 10000/10000 [==============================] - 15s 1ms/step

[0.8423407137393951, 0.7155999958515167]
```

**Рис. 2.18.** Результаты классификации с использованием CNN

**Рис. 2.19.** Прогнозы, сделанные сетью CNN

Столь заметное улучшение достигнуто простым изменением архитектуры модели, включением в нее сверточных слоев, пакетной нормализации и прореживания. Обратите внимание: количество параметров в новой модели меньше, чем в предыдущей, хотя количество слоев и увеличилось. Это доказывает, насколько важно экспериментировать с архитектурой модели и понимать, как в своих интересах использовать разные типы слоев. При конструировании генеративных моделей еще важнее понимать внутреннюю работу модели, потому что именно слои в середине сети выявляют высокоуровневые признаки, которые больше всего нас интересуют.

## Итоги

Итак, мы рассмотрели основные идеи глубокого обучения, которые понадобятся вам при создании своих первых глубоких генеративных моделей. Самый важный урок этой главы — в том, что глубокие нейронные сети по своей природе обладают большой гибкостью и не существует правил, жестко регламентирующих создание архитектуры модели, что позволяет смело экспериментировать со слоями и их порядком. Кроме того, некоторые слои, как и набор строительных блоков, могут оказаться несовместимыми друг с другом, поскольку форма входных данных одного слоя не соответствует форме выходных данных другого. Эти знания придут с опытом и глубоким пониманием того, как каждый слой меняет форму тензора при передаче данных по сети.

Важно и то, что сверточными являются *слои* в глубокой нейронной сети, а не сама сеть. Соответственно, под «сверточными нейронными сетями» в действительности подразумеваются «нейронные сети, содержащие сверточные слои». Важно помнить об этом, чтобы не чувствовать себя обязанными использовать только архитектуры, заимствованные из книг, — рассматривайте их как примеры, демонстрирующие возможность сочетания слоев разных типов. Архитектура создаваемой нейронной сети ограничена только вашим воображением и, главное, пониманием того, как различные слои сочетаются друг с другом. Далее мы увидим, как использовать эти строительные блоки для конструирования сети, способной генерировать изображения.

ГЛАВА 3

# Вариационные автокодировщики

В 2013 году Дидерик П. Кингма (Diederik P. Kingma) и Макс Веллинг (Max Welling) опубликовали статью, заложившую основы типа нейронной сети, известной как вариационный автокодировщик (variational autoencoder, VAE)[1]. Теперь это одна из самых фундаментальных и хорошо известных архитектур глубокого обучения для генеративного моделирования. Эту главу мы и начнем с создания стандартного автокодировщика, а затем попытаемся получить вариационный автокодировщик — наш первый пример генеративной модели глубокого обучения. Попутно детально рассмотрим оба типа моделей, чтобы понять, как они работают. К концу главы вы получите полное представление о том, как создавать модели и манипулировать ими на основе автокодировщика и, в частности, как создавать вариационный автокодировщик с нуля для генерации изображений на основе вашего собственного обучающего набора.

Начнем же мы с посещения необычной художественной выставки.

## Художественная выставка

Два брата, мистер Н. Кодер и мистер Д. Кодер, руководят художественной галереей. По выходным здесь проводится выставка, посвященная исследованиям монохромных изображений одноразрядных чисел (состоящих из единственной цифры). Выставка необычна тем, что размещена только на

[1] Дидерик П. Кингма (Diederik P. Kingma) и Макс Веллинг (Max Welling), «Auto-Encoding Variational Bayes», 20 декабря 2013, https://arxiv.org/abs/1312.6114.

одной стене (рис. 3.1) и не содержит фактических работ. Когда появляется новая картина для показа, мистер Н. Кодер просто выбирает место на стене для картины, ставит маркером точку в этом месте, а затем выбрасывает оригинальную работу. Когда клиент просит показать картину, мистер Д. Кодер пытается воссоздать ее, маркируя только координаты точки на стене. Каждая черная точка на ней — это след маркера, оставленный мистером Н. Кодером для представления картины. Здесь же показана одна из картин, отмеченная мистером Н. Кодером на стене точкой с координатами `[-3.5, -0.5]` и реконструированная мистером Д. Кодером на основе этих чисел.

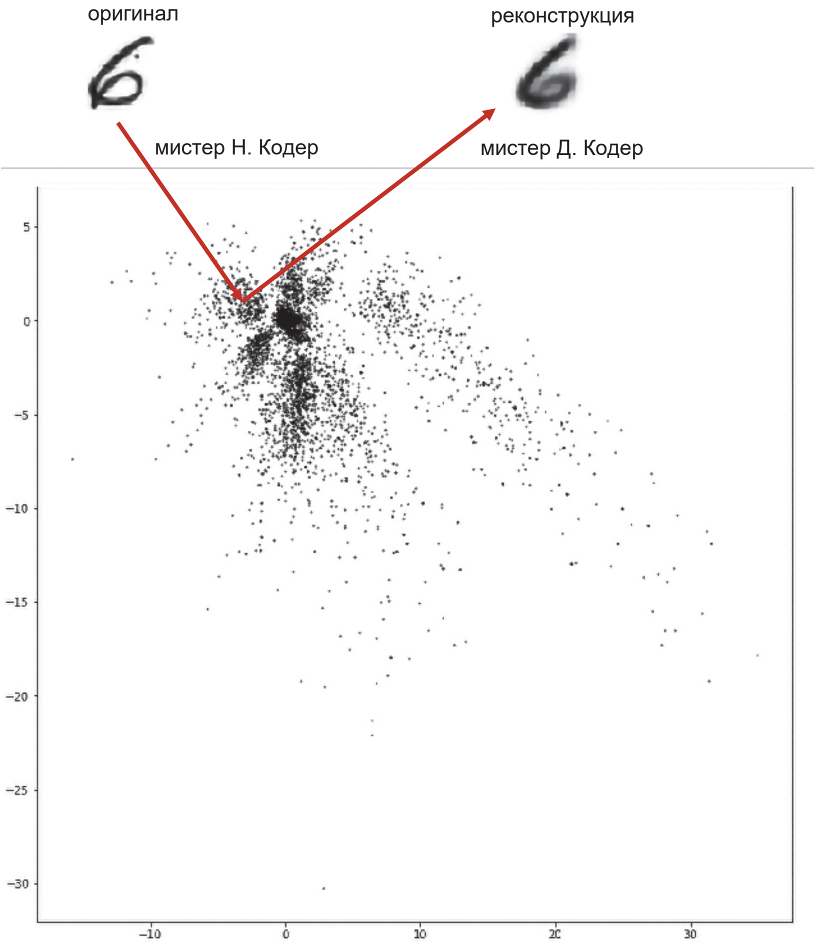



**Рис. 3.1.** Стена для художественной выставки

На рис. 3.2 представлены примеры других оригинальных картин (верхний ряд), координаты точек на стене, поставленных мистером Н. Кодером, и картины, воссозданные мистером Д. Кодером (нижний ряд).

Но как мистер Н. Кодер выбирает место для маркировки? Система развивалась в течение нескольких лет обучения и совместной работы. Постепенно изменялись критерии размещения точек и реконструкции картин по ним. Братья внимательно следят за потерей доходов, когда посетители требуют вернуть деньги из-за плохо реконструированных картин, постоянно совершенствуют систему, сводя эти потери к минимуму. Как видно на рис. 3.2, система работает замечательно — клиенты очень редко жалуются на то, что картины, воссозданные мистером Д. Кодером, значительно отличаются от оригинальных произведений, которые они видели ранее.

Однажды у мистера Н. Кодера появилась идея. А что, если случайно поставить точки в тех местах на стене, где в настоящее время отсутствуют любые точки? Затем мистер Д. Кодер мог бы попробовать воссоздать картины, соответствующие этим точкам, и за несколько дней они смогли бы получить выставку совершенно оригинальных произведений. Недолго думая, братья приступили к реализации своего плана и уже вскоре открыли новую выставку для публики (рис. 3.3).

Как видим, попытка не имела большого успеха. Общее разнообразие оставляет желать лучшего, причем некоторые произведения не очень похожи на одноразрядные числа. Попробуем разобраться, что пошло не так и как братья могли бы усовершенствовать свою схему.

**Рис. 3.2.** Еще примеры оригинальных картин и их реконструкций

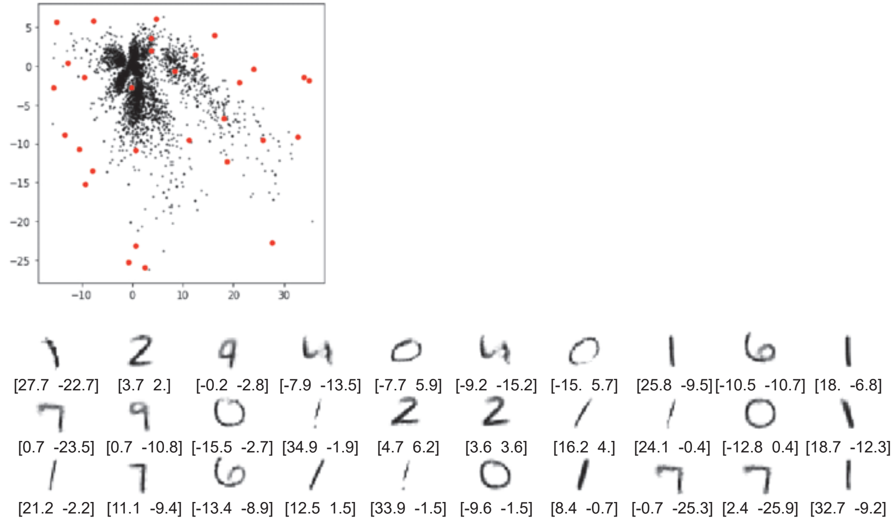



**Рис. 3.3.** Новая выставка генеративного искусства

## Автокодировщики

Этот рассказ описывает аналог автокодировщика – двухкомпонентной нейронной сети, включающей (рис. 3.4):

- сеть *кодировщика*, которая сжимает входные многомерные данные в вектор меньшего размера;
- сеть *декодировщика*, которая разворачивает данный вектор представления обратно в исходные данные.

В процессе обучения сеть определяет веса для кодировщика и декодировщика, минимизирующие потери между оригинальными входными и реконструированными данными, полученными после прохождения входных данных через кодировщик и декодировщик. Вектор представления – это результат сжатия исходного изображения в скрытое пространство с меньшим числом измерений. Идея в том, что при выборе *любой* точки в скрытом пространстве мы должны получить новое изображение, пропустив эту точку через декодировщик, обученный преобразованию точки в скрытом пространстве в соответствующее изображение.

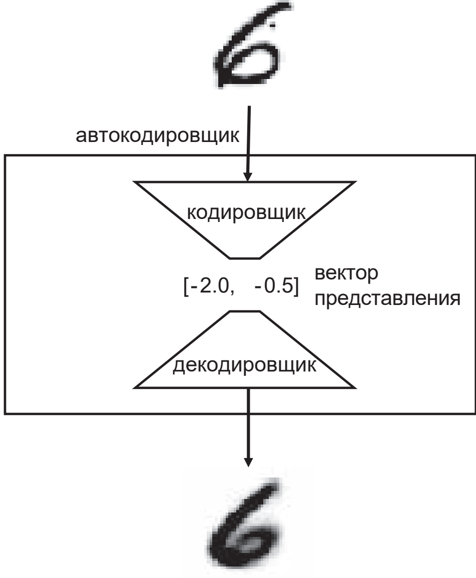



**Рис. 3.4.** Схема кодирования/декодирования

В рассказе (см. выше) для кодирования каждого изображения мистер Н. Кодер и мистер Д. Кодер используют векторы представления в двумерном скрытом пространстве (на стене). Эта аналогия помогает нам представить скрытое пространство, потому что мы легко можем ставить точки на двумерной плоскости. На практике автокодировщики обычно имеют больше двух измерений, чтобы иметь больше свободы для выявления большего числа нюансов в изображениях.

Автокодировщики также можно использовать для очистки искаженных изображений от шума, потому что в процессе обучения кодировщик узнает, что бессмысленно пытаться фиксировать случайный шум в скрытом пространстве. В подобных задачах двумерное скрытое пространство слишком мало для представления релевантной информации, присутствующей во входных данных. Однако, как мы увидим далее, увеличение размерности скрытого пространства быстро приводит к проблемам, если использовать автокодировщик в качестве генеративной модели.

## Ваш первый автокодировщик

Теперь сконструируем автокодировщик с помощью Keras. Его реализацию вы найдете в блокноте 03_01_autoencoder_train.ipynb Jupyter Notebook (в репозитории с примерами для книги). Вообще рекомендуется создавать классы моделей в отдельных файлах. При таком подходе вы получаете возможность создавать экземпляры объекта `Autoencoder` с параметрами, определяющими конкретную архитектуру модели в блокноте (листинг 3.1). Это сделает модель очень гибкой, простой в тестировании и легко переносимой в другие проекты, если это потребуется.

**Листинг 3.1.** Определение автокодировщика

```
from models.AE import Autoencoder

AE = Autoencoder(
    input_dim = (28,28,1)
    , encoder_conv_filters = [32,64,64, 64]
    , encoder_conv_kernel_size = [3,3,3,3]
    , encoder_conv_strides = [1,2,2,1]
    , decoder_conv_t_filters = [64,64,32,1]
    , decoder_conv_t_kernel_size = [3,3,3,3]
    , decoder_conv_t_strides = [1,2,2,1]
    , z_dim = 2)
```

Рассмотрим архитектуру автокодировщика подробнее, начав, однако, с кодировщика.

## Кодировщик

Задача кодировщика – отобразить входное изображение в точку в скрытом пространстве.

Чтобы получить требуемую архитектуру (рис. 3.5), создадим входной слой для приема изображений, последовательно пропустив их через четыре слоя `Conv2D`, каждый из которых выявляет все более высокоуровневые элементы. Используем размер шага 2 в некоторых слоях, чтобы уменьшить размер выходных данных. Результат последнего сверточного слоя преобразуется в вектор и передается в слой `Dense` с размером 2, представляющий наше двумерное скрытое пространство (листинг 3.2).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   (None, 28, 28, 1)         0
_________________________________________________________________
encoder_conv_0 (Conv2D)      (None, 28, 28, 32)        320
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 28, 28, 32)        0
_________________________________________________________________
encoder_conv_1 (Conv2D)      (None, 14, 14, 64)        18496
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
encoder_conv_2 (Conv2D)      (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
encoder_conv_3 (Conv2D)      (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
encoder_output (Dense)       (None, 2)                 6274
=================================================================
Total params: 98,946
Trainable params: 98,946
Non-trainable params: 0
_________________________________________________________________
```

**Рис. 3.5.** Архитектура кодировщика

**Листинг 3.2.** Кодировщик

```
### КОДИРОВЩИК
encoder_input = Input(shape=self.input_dim, name='encoder_input') ❶

x = encoder_input

for i in range(self.n_layers_encoder):
    conv_layer = Conv2D(
        filters = self.encoder_conv_filters[i]
        , kernel_size = self.encoder_conv_kernel_size[i]
        , strides = self.encoder_conv_strides[i]
        , padding = 'same'
        , name = 'encoder_conv_' + str(i)
        )

    x = conv_layer(x) ❷
```

```
    x = LeakyReLU()(x)

shape_before_flattening = K.int_shape(x)[1:]
x = Flatten()(x) ❸

encoder_output= Dense(self.z_dim, name='encoder_output')(x) ❹

self.encoder = Model(encoder_input, encoder_output) ❺
```

❶ Определение входного слоя кодировщика (принимающего изображение).

❷ Стопка сверточных слоев, наложенных друг на друга.

❸ Преобразование результата последнего сверточного слоя в плоский вектор.

❹ Слой `Dense`, преобразующий вектор в двумерное скрытое пространство.

❺ Модель Keras, определяющая кодировщик, — модель, которая принимает исходное изображение и кодирует его в точку в двумерном пространстве.

Чтобы изменить количество сверточных слоев в кодировщике, достаточно добавить новые элементы в списки, определяющие архитектуру модели в блокноте. Обязательно экспериментируйте с параметрами моделей, представленных в этой книге, чтобы понять, как архитектура влияет на количество весов в каждом слое, качество и время выполнения модели.

## Декодировщик

Декодировщик является зеркальным отражением кодировщика, но вместо сверточных слоев в нем используются слои *обратной свертки* (рис. 3.6).

Отметим, архитектура декодировщика необязательно должна быть зеркальным отражением архитектуры кодировщика. Она может быть какой угодно, если выход последнего слоя декодировщика имеет тот же размер, что и вход кодировщика (потому что наша функция потерь будет сравнивать их по пикселам).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   (None, 2)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 3136)              9408
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 64)          0
_________________________________________________________________
decoder_conv_t_0 (Conv2DTran (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
decoder_conv_t_1 (Conv2DTran (None, 14, 14, 64)        36928
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
decoder_conv_t_2 (Conv2DTran (None, 28, 28, 32)        18464
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 28, 28, 32)        0
_________________________________________________________________
decoder_conv_t_3 (Conv2DTran (None, 28, 28, 1)         289
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 1)         0
=================================================================
Total params: 102,017
Trainable params: 102,017
Non-trainable params: 0
_________________________________________________________________
```

**Рис. 3.6.** Архитектура декодировщика

Листинг 3.3 демонстрирует реализацию декодировщика с использованием Keras.

**Листинг 3.3.** Декодировщик

```
### ДЕКОДИРОВЩИК
decoder_input = Input(shape=(self.z_dim,), name='decoder_input') ❶

x = Dense(np.prod(shape_before_flattening))(decoder_input) ❷
x = Reshape(shape_before_flattening)(x) ❸

for i in range(self.n_layers_decoder):
    conv_t_layer = Conv2DTranspose(
        filters = self.decoder_conv_t_filters[i]
        , kernel_size = self.decoder_conv_t_kernel_size[i]
        , strides = self.decoder_conv_t_strides[i]
        , padding = 'same'
        , name = 'decoder_conv_t_' + str(i)
```

```
        )

    x = conv_t_layer(x) ❹

    if i < self.n_layers_decoder - 1:
        x = LeakyReLU()(x)
    else:
        x = Activation('sigmoid')(x)

decoder_output = x

self.decoder = Model(decoder_input, decoder_output) ❺
```

### СЛОИ ОБРАТНОЙ СВЕРТКИ

Обычный сверточный слой позволяет вдвое уменьшить высоту и ширину входного тензора, если установить параметр `strides = 2`. Слой обратной свертки использует тот же принцип, что и обычный сверточный (применяет фильтр к изображению), с той лишь разницей, что параметр `strides = 2` *удваивает* высоту и ширину входного тензора. В слое обратной свертки параметр `strides` определяет размер внутреннего дополнения для добавления нулевых значений между пикселами в изображении (рис. 3.7).

Слой `Conv2DTranspose` в Keras позволяет применять к тензорам операцию, обратную свертке. Добавляя такие слои, можно постепенно увеличивать размер выходных данных, используя параметр `strides = 2`, до получения размеров, соответствующих размерам оригинального изображения 28 × 28.

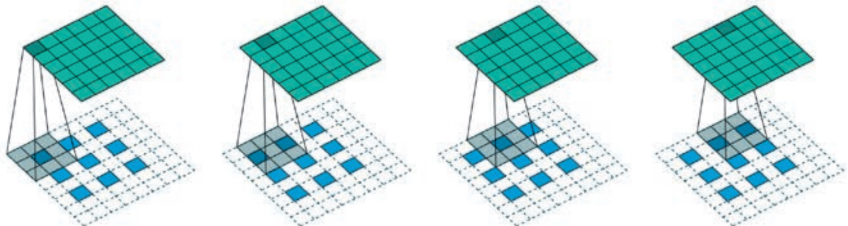


**Рис. 3.7.** Принцип работы слоя обратной свертки: здесь фильтр 3 × 3 × 1 (серый) последовательно накладывается на изображение 3 × 3 × 1 (синее) с параметром strides = 2, и в результате получается выходной тензор 6 × 6 × 1 (зеленый)[1]

[1] Винсент Дюмолин (Vincent Dumoulin) и Франческо Висин (Francesco Visin), «A Guide to Convolution Arithmetic for Deep Learning», 12 January 2018, https://arxiv.org/pdf/1603.07285.pdf.

❶ Определение входного слоя декодировщика (принимающего точку в скрытом пространстве).

❷ Входные данные передаются в слой `Dense`.

❸ Полученный вектор преобразуется в тензор с формой, в которой его можно передать первому слою обратной свертки.

❹ Стопка слоев обратной свертки, наложенных друг на друга.

❺ Модель Keras, определяющая декодировщик, — модель, принимающая точку в скрытом пространстве и декодирующая ее в изображение в исходном пространстве.

## Объединение кодировщика и декодировщика

Чтобы обучить кодировщик и декодировщик одновременно, нужно определить модель, которая будет направлять поток изображений через кодировщик и обратно через декодировщик. К счастью, Keras упрощает эту задачу (листинг 3.4).

**Листинг 3.4.** Полный автокодировщик

```
### ПОЛНЫЙ АВТОКОДИРОВЩИК
model_input = encoder_input # ❶
model_output = decoder(encoder_output) # ❷

self.model = Model(model_input, model_output) # ❸
```

❶ На вход автокодировщика подаются те же данные, что и на вход кодировщика.

❷ Выходные данные автокодировщика — это выходные данные кодировщика, прошедшие через декодировщик.

❸ Модель Keras, определяющая полный автокодировщик, — модель, которая принимает изображение, пропускает его через кодировщик и затем обратно через декодировщик, в результате чего получается реконструированное изображение.

Теперь, когда у нас есть модель, осталось только скомпилировать ее с функцией потерь и оптимизатором (листинг 3.5). На роль функции потерь обычно выбирается либо среднеквадратичная ошибка (Root Mean Squared

Error, RMSE), либо бинарная перекрестная энтропия между отдельными пикселами исходного изображения и его реконструкции. Бинарная перекрестная энтропия накладывает более жесткие штрафы на предсказания, сильно отличающиеся от истины, что способствует сдвигу предсказаний в середину диапазона. В результате сгенерированные изображения получаются менее эффектными. По этой причине предпочтительнее использовать RMSE. Но имейте в виду, что нет заведомо правильного или неправильного выбора — желательно попробовать все варианты и выбрать тот, который лучше подходит для вашего случая.

**Листинг 3.5.** Компиляция

```
### КОМПИЛЯЦИЯ
optimizer = Adam(lr=learning_rate)

def r_loss(y_true, y_pred):
    return K.mean(K.square(y_true - y_pred), axis = [1,2,3])

self.model.compile(optimizer=optimizer, loss = r_loss)
```

Обучим модель автокодировщика, передав исходные изображения на вход и выход (листинг 3.6).

**Листинг 3.6.** Обучение модели автокодировщика

```
self.model.fit(
    x = x_train
    , y = x_train
    , batch_size = batch_size
    , shuffle = True
    , epochs = 10
    , callbacks = callbacks_list
)
```

## Анализ автокодировщика

Завершив обучение, посмотрим, как автокодировщик представляет изображения в скрытом пространстве. Затем мы познакомимся с вариационными автокодировщиками — естественным расширением, устраняющим проблемы, характерные для обычных автокодировщиков (см. код в блокноте 03_02_autoencoder_analysis.ipynb в составе репозитория с примерами для книги).

Возьмем набор новых изображений, которые модель не видела в процессе обучения, пропустим их через кодировщик и нанесем их двумерные представления на диаграмму рассеяния. На самом деле мы уже видели эту диаграмму на рис. 3.1: это стена, на которой мистер Н. Кодер ставил точки. Раскрасив точки в зависимости от цифр, которые они представляют, получаем диаграмму, изображенную на рис. 3.8. Стоит отметить, что хотя во время обучения модель не видела меток, указывающих на цифры, автокодировщик сумел сгруппировать похожие по начертанию цифры в одну область скрытого пространства.

Вот несколько интересных наблюдений, которые можно сделать, рассматривая эту диаграмму:

1. Точки расположены несимметрично относительно (0, 0), и области их концентрации не имеют четких границ. Например, точек с отрицательным значением координаты $y$ намного больше, чем с положительным, а некоторые даже имеют координату $y < -30$.
2. Точки, соответствующие некоторым цифрам, концентрируются в очень маленькой области, а другие более рассеяны на координатной плоскости.
3. Имеются большие пустые области между группами точек разного цвета.

Напомним, наша цель — в том, чтобы получить возможность выбрать случайную точку в скрытом пространстве, пропустить ее через декодировщик и получить изображение цифры, которая похожа на настоящую. В идеале желательно, чтобы после нескольких таких попыток у нас получилась коллекция, содержащая примерно равное количество разных типов цифр (то есть модель не должна генерировать одну и ту же цифру). Эту же цель преследовали и братья Кодер, когда выбирали случайные точки на своей стене для создания нового изображения для своей выставки.

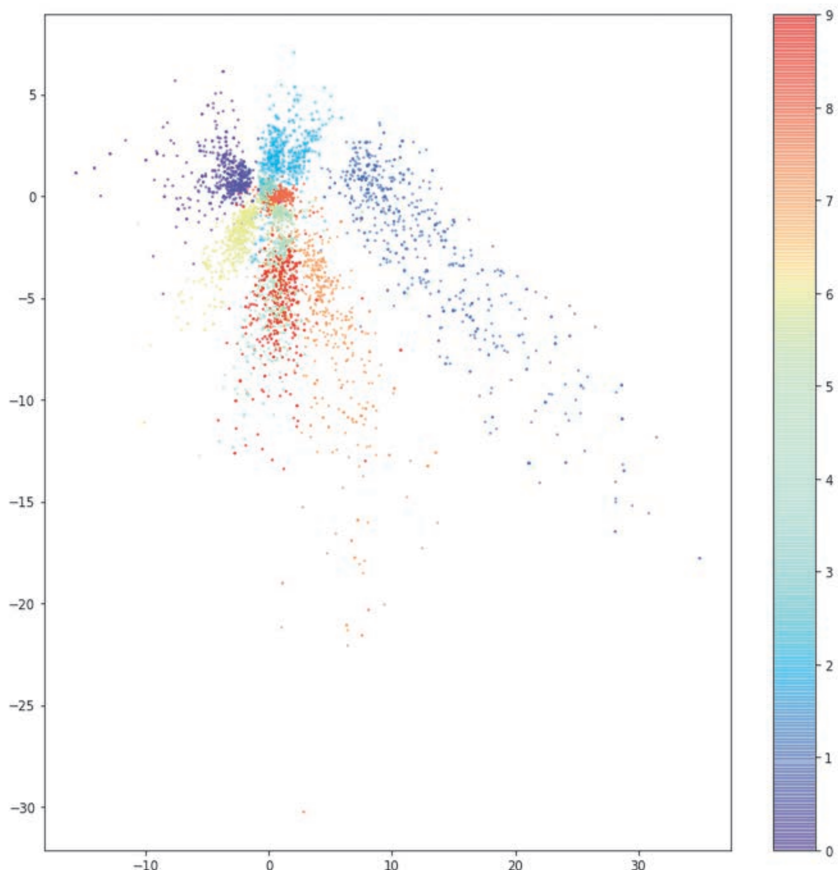



**Рис. 3.8.** Скрытое пространство, в котором точки, соответствующие разным цифрам, раскрашены в разные цвета

Пункт 1 объясняет неочевидность правил, которыми желательно руководствоваться при выборе *случайной* точки в скрытом пространстве, неизвестным распределением точек. Технически мы вольны выбрать любую точку на плоскости, причем не гарантируется, что точки будут сосредоточены вокруг (0,0), что делает выбор представительного образца из скрытого пространства чрезвычайно проблематичной задачей.

Пункт 2 объясняет отсутствие разнообразия в сгенерированных изображениях. В идеале, выбирая случайные точки в скрытом пространстве, мы хотели бы получить примерно равное количество разных цифр. Но авто-

кодировщик не гарантирует этого. Например, область цифры «1» намного больше, чем область цифры «8», и, выбирая случайные точки в скрытом пространстве, мы с большей вероятностью выберем то, что декодируется в изображение, больше похожее на «1», а не на «8».

Пункт 3 объясняет, почему некоторые сгенерированные изображения мало похожи на цифры. На рис. 3.9 показаны три точки в скрытом пространстве и соответствующие им декодированные изображения, ни одно из которых не является четко сформированным изображением цифры.

Отчасти это обусловлено наличием обширных пустых областей на границах пространств, где присутствует всего несколько точек, — у автокодировщика нет оснований гарантировать декодирование этих точек в разборчивые цифры, поскольку в эти области кодируется очень мало изображений. Однако, что еще более неприятно, даже точки, находящиеся в середине пространства, могут не декодироваться в четкие изображения. Это связано с тем, что автокодировщик не обязан обеспечивать *непрерывность* пространства. Например, точка (2, –2) может декодироваться во вполне удовлетворительное изображение «4», а точка (2.1, –2.1) уже может не давать столь же удовлетворительного изображения «4» из-за отсутствия необходимых для этого механизмов.

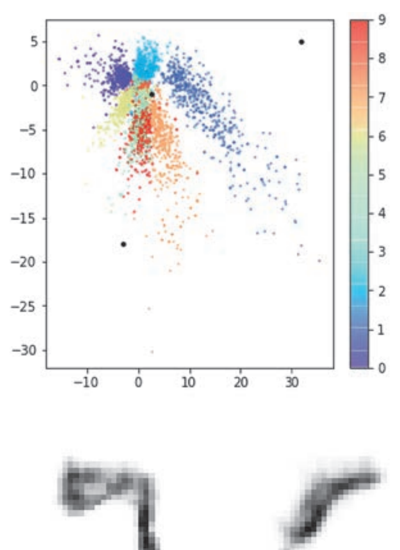



**Рис. 3.9.** Некоторые некачественно сгенерированные изображения

При использовании двумерного скрытого пространства эта проблема малозаметна – автокодировщик ограничен небольшим количеством измерений и вынужден втискивать группы цифр в узкие рамки. В результате пространство между этими группами оказывается относительно невелико. Но с увеличением числа измерений для создания более сложных изображений, в том числе и таких, как лица, проблема становится более очевидной. Если дать автокодировщику полную свободу действий в использовании скрытого пространства для кодирования изображений, то мы получим огромные промежутки между группами похожих точек, не позволяющие получить правильно сформированные изображения. Как же можно решить эти проблемы, подготовив автокодировщик к использованию в роли генеративной модели? Вернемся к художественной выставке братьев Кодер, где с момента нашего последнего визита произошли некоторые изменения...

## Выставка вариационного искусства

Мистер Н. Кодер был полон решимости сделать выставку генеративного искусства высокодоходной и обратился за помощью к своей дочери Эпсилон. После краткого обсуждения они решили изменить способ выбора точек на стене, представляющих картины, следующим образом. Когда новая картина прибывает на выставку, мистер Н. Кодер выбирает на стене место, где хотел бы поставить точку, представляющую изображение, как и раньше. Но теперь он не ставит точку сразу же, как только выберет место для нее, а сообщает свое мнение дочери Эпсилон, которая уточняет местоположение точки. Конечно, она учитывает мнение своего отца, поэтому обычно ставит точку где-то рядом с местом, которое предложил отец. Но мистер Д. Кодер в своей работе использует только точки, поставленные Эпсилон, ничего не зная о первоначальном мнении мистера Н. Кодера.

Мистер Н. Кодер также сообщает своей дочери, насколько он уверен, что точка должна находиться именно в этом месте. Чем более он уверен, тем ближе к предложенному месту Эпсилон ставит окончательную точку.

И еще одно изменение в старой системе: раньше единственной обратной связью была потеря дохода из-за некачественной реконструкции изображений. Если братья видели, что какие-то изображения воссозданы недостаточно точно, они могли скорректировать местоположение точки и вновь сгенерировать изображение, чтобы уменьшить потери доходов.

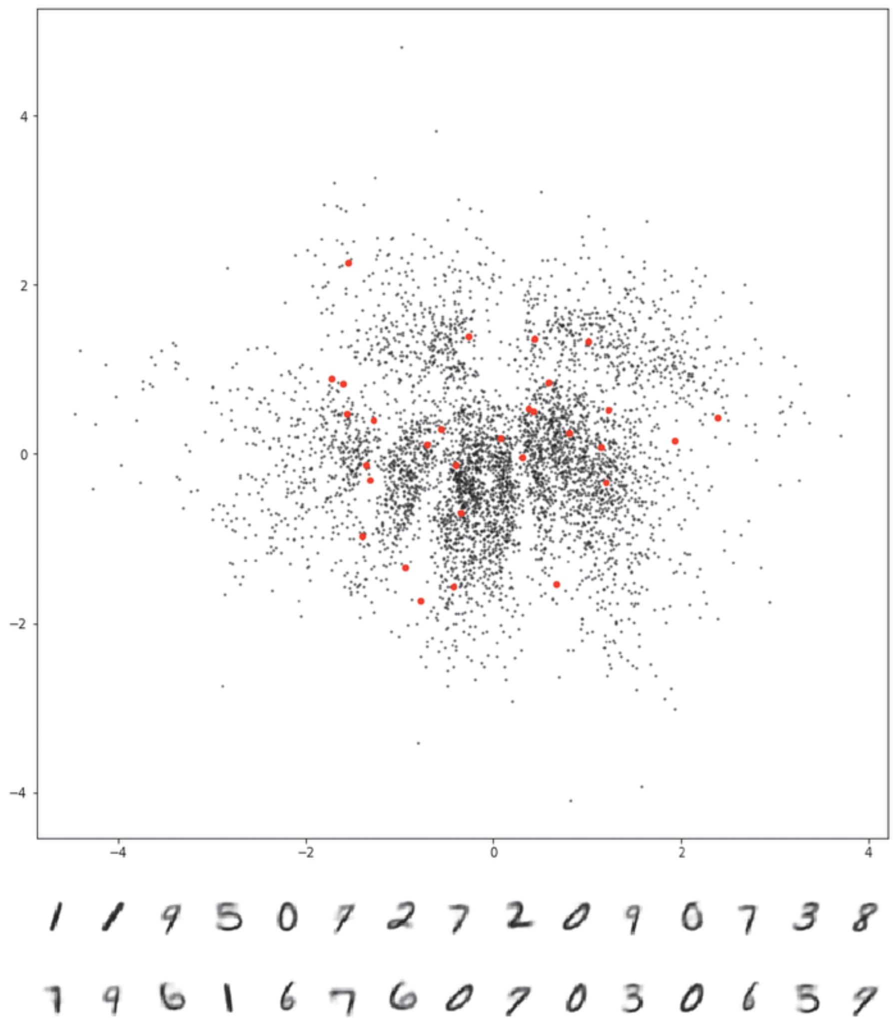



**Рис. 3.10.** Картины с новой выставки

Теперь появился еще один источник обратной связи. Однако Эпсилон довольно ленива и ее раздражает, когда отец предлагает поставить точку подальше от центра стены, где лежит лестница. Ей также не нравится, когда он слишком строго определяет местоположение точек, поскольку чувствует ограничение своей свободы. Точно так же, если отец не очень уверен, куда поставить точку, у нее возникает ощущение, что именно она делает всю

работу! Его уверенность в местоположении точек должна давать Эпсилон ощущение счастья от выполненной работы.

Чтобы компенсировать недовольство дочери, отец платит ей за работу больше, если не придерживается этих правил. В бухгалтерской отчетности эти расходы указаны как конфиденциальные. Поэтому он должен быть осторожным, чтобы не платить слишком много своей дочери, а также следить за потерей дохода в кассе. После этих простых изменений мистер Н. Кодер вновь пробует применить свою стратегию размещения точек на пустых участках стены, чтобы мистер Д. Кодер мог восстановить их в оригинальные картины (рис. 3.10).

В результате в галерею повалили толпы посетителей, желающих увидеть это новое, захватывающее генеративное искусство и удивиться оригинальности и разнообразию картин.

## Конструирование вариационного автокодировщика

Предыдущая история показала, как несколькими простыми изменениями превратить художественную выставку в успешный генеративный процесс. Теперь попробуем разобраться с математической точки зрения, что нужно сделать с нашим автокодировщиком, чтобы преобразовать его в вариационный автокодировщик и превратить в настоящую генеративную модель. На самом деле изменить нужно только две части: кодировщик и функцию потерь.

### Кодировщик

В автокодировщике каждое изображение отображается непосредственно в точку в скрытом пространстве. В вариационном автокодировщике каждое изображение отображается в многомерное нормальное распределение вокруг точки в скрытом пространстве (рис. 3.11).

Вариационные автокодировщики предполагают отсутствие любой корреляции между измерениями скрытого пространства и, следовательно, то, что ковариационная матрица является диагональной. Это означает, что кодировщик должен лишь отобразить каждый вход в вектор средних и вектор дисперсии, не беспокоясь о ковариации между измерениями. Часто также выбирается отображение в *логарифм* дисперсии, потому что он может принимать любые действительные значения в диапазоне ($-\infty$, $\infty$), что соответ-

ствует естественному диапазону выходных значений узла нейронной сети, тогда как значения дисперсии всегда положительны. В итоге кодировщик будет преобразовывать каждое входное изображение в два вектора, `mu` и `log_var`, которые вместе определяют многомерное нормальное распределение в скрытом пространстве:

`mu`

Средняя точка распределения.

`log_var`

Логарифм дисперсии каждого измерения.

Преобразовать изображение в определенную точку `z` в скрытом пространстве можем путем выборки из этого распределения с помощью следующего уравнения:

```
z = mu + sigma * epsilon,
```

где `sigma` вычисляется так[1]:

```
sigma = exp(log_var / 2)
```

`epsilon` — это точка, выбранная из стандартного нормального распределения.

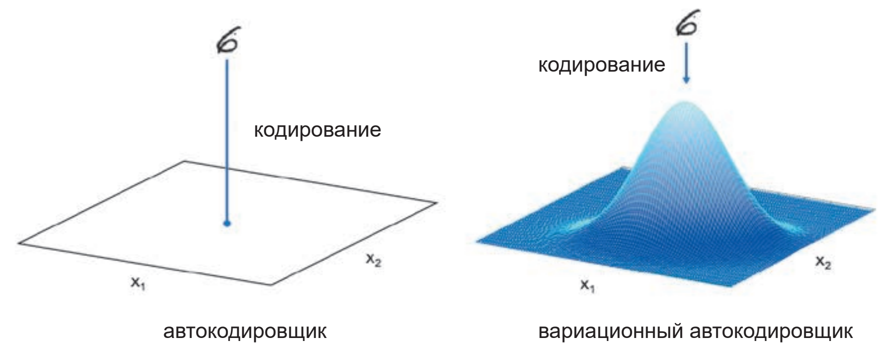



**Рис. 3.11.** Различия между кодировщиком в обычном и вариационном автокодировщике

[1] $\sigma = \exp(\log(\sigma)) = \exp(2\log(\sigma^2)/2)$.

**НОРМАЛЬНОЕ РАСПРЕДЕЛЕНИЕ**

Нормальное распределение — это распределение вероятностей, характеризующееся специфической колоколообразной формой кривой. Одномерное распределение определяется двумя переменными: *средним* ($\mu$) и дисперсией ($\sigma^2$). *Стандартное отклонение* ($\sigma$) — это квадратный корень из дисперсии.

Функция плотности вероятности одномерного нормального распределения имеет вид

$$f\left(x|\mu,\sigma^2\right)=\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

На рис. 3.12 показано несколько одномерных нормальных распределений с разными значениями среднего и дисперсии. Красная кривая соответствует *стандартному нормальному распределению* — нормальному распределению со средним значением 0 и дисперсией 1.

Выбрать точку z из нормального распределения со средним $\mu$ и стандартным отклонением $\sigma$ можно с помощью уравнения

$$z = \mu + \sigma\epsilon,$$

где $\epsilon$ — значение, выбранное из стандартного нормального распределения.

Нормальное распределение может быть не только одномерным — функция плотности вероятности для обобщенного многомерного нормального распределения с $k$ измерениями выглядит так:

$$f(x_1,\dots x_k)=\frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\sum^{-1}(\mathbf{x}-\mu)\right)}{\sqrt{(2\pi)^k|\Sigma|}}.$$

Для двумерного распределения вектор средних $\mu$ и симметричная ковариационная матрица $\Sigma$ определяются так:

$$\mu=\begin{pmatrix}\mu_1\\ \mu_2\end{pmatrix},\qquad \Sigma=\begin{pmatrix}\sigma_1^2 & \rho\sigma_1\sigma_2\\ \rho\sigma_1\sigma_2 & \sigma_2^2\end{pmatrix},$$

где $\rho$ — коэффициент корреляции между двумя измерениями $x_1$ и $x_2$.

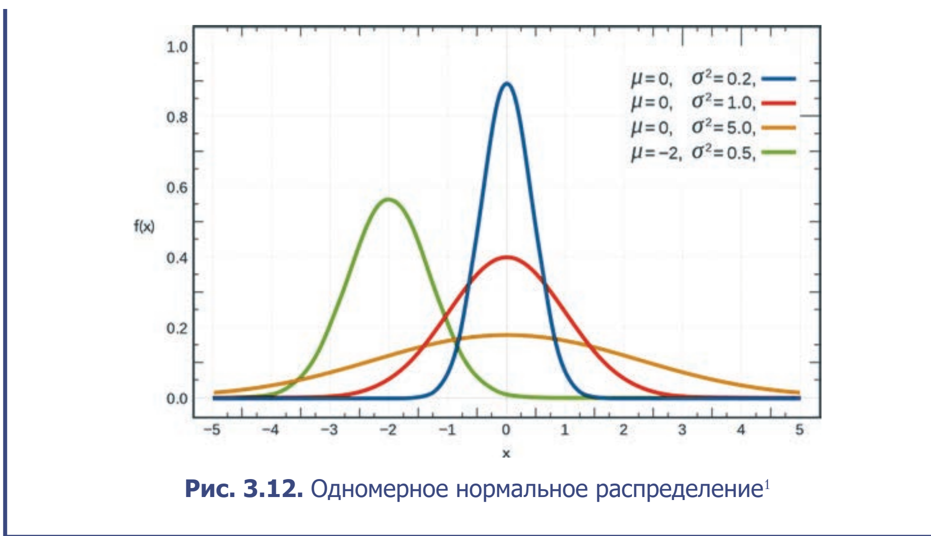



**Рис. 3.12.** Одномерное нормальное распределение[1]

В нашей истории `mu` представляет мнение мистера Н. Кодера о том, где на стене должна появиться точка. `epsilon` — это случайный выбор его дочери, определяющий расстояние от `mu`, где фактически следует поставить точку, масштабированное значением `sigma` — уверенностью мистера Н. Кодера в местоположении точки. Но почему столь небольшое изменение способствует повышению качества кодирования?

Прежде к непрерывному пространству не предъявлялось требование непрерывности — даже если точка (–2, 2) декодировалась в правильно сформированное изображение цифры «4», это не означало, что изображение для точки (–2.1, 2.1) должно быть похожим. Теперь же, поскольку мы выбираем случайную точку из области вокруг `mu`, декодировщик должен декодировать все соседние точки в очень похожие изображения, то есть потери при реконструкции должны оставаться небольшими. Это свойство гарантирует, что при выборе точки в скрытом пространстве, которую декодировщик прежде никогда не видел, она с большой вероятностью будет декодирована в хорошо сформированное изображение. Посмотрим, как сконструировать эту версию кодировщика в Keras (листинг 3.7). Вы можете обучить собственный вариационный автокодировщик, запустив блокнот 03_03_vae_digits_train.ipynb из репозитория с примерами для книги.

[1] Источник: Википедия, https://ru.wikipedia.org/wiki/Нормальное_распределение.

**Листинг 3.7.** Кодировщик вариационного автокодировщика

```
### КОДИРОВЩИК
encoder_input = Input(shape=self.input_dim, name='encoder_input')

x = encoder_input

for i in range(self.n_layers_encoder):
    conv_layer = Conv2D(
        filters = self.encoder_conv_filters[i]
        , kernel_size = self.encoder_conv_kernel_size[i]
        , strides = self.encoder_conv_strides[i]
        , padding = 'same'
        , name = 'encoder_conv_' + str(i)
        )

    x = conv_layer(x)

    if self.use_batch_norm:
        x = BatchNormalization()(x)

    x = LeakyReLU()(x)
    if self.use_dropout:
        x = Dropout(rate = 0.25)(x)

shape_before_flattening = K.int_shape(x)[1:]
x = Flatten()(x)

self.mu = Dense(self.z_dim, name='mu')(x) ❶
self.log_var = Dense(self.z_dim, name='log_var')(x) #

encoder_mu_log_var = Model(encoder_input, (self.mu, self.log_var)) ❷

def sampling(args):
    mu, log_var = args
    epsilon = K.random_normal(shape=K.shape(mu), mean=0., stddev=1.)
    return mu + K.exp(log_var / 2) * epsilon

encoder_output = Lambda(sampling, name='encoder_output')([self.mu, self.log_
                                                             var]) ❸

encoder = Model(encoder_input, encoder_output) ❹
```

❶ Слой преобразования в плоский вектор подключается не к двумерному скрытому пространству непосредственно, а к слоям `mu` и `log_var`.

❷ Модель Keras, которая возвращает значения `mu` и `log_var` для данного входного изображения.

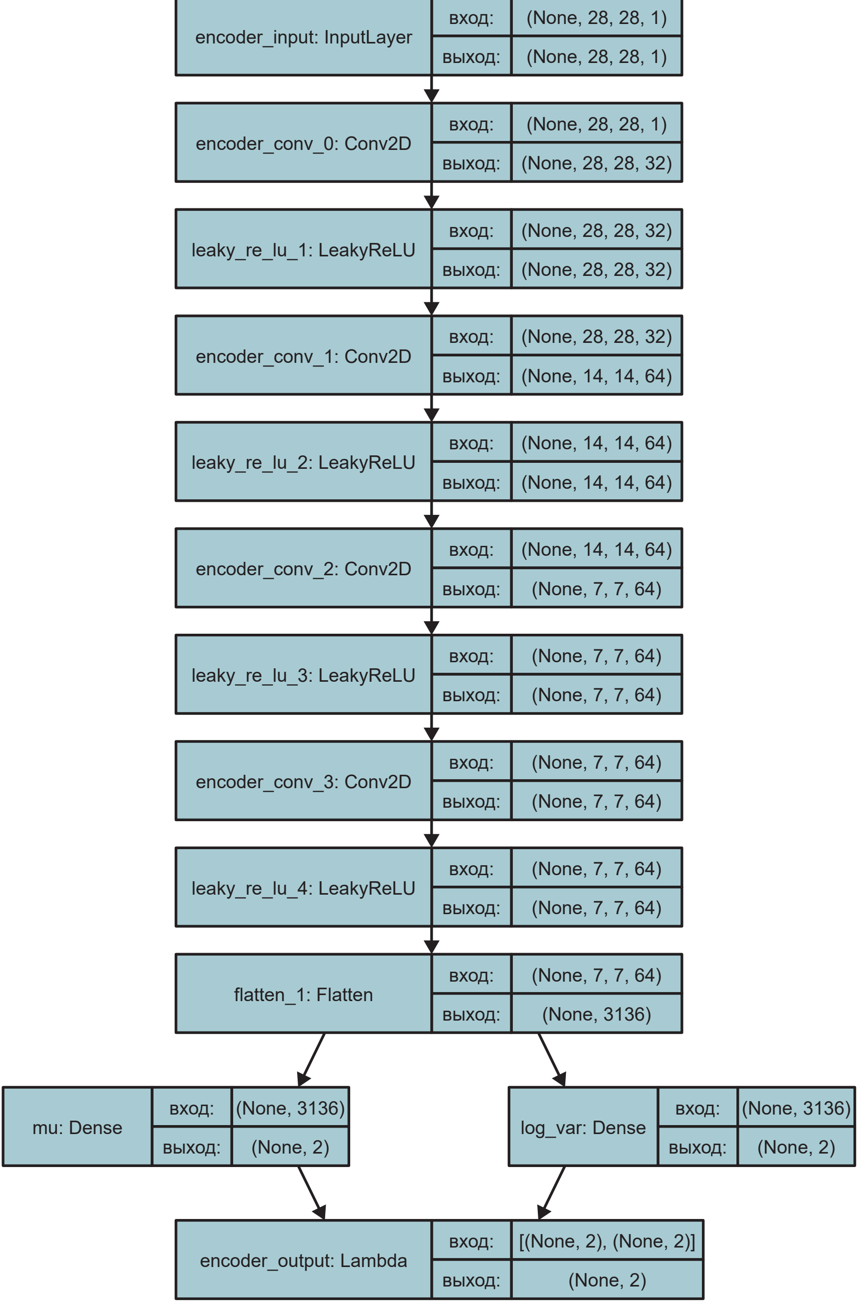



**Рис. 3.13.** Схема с устройством кодировщика в вариационном автокодировщике

❸ Этот слой `Lambda` выбирает представление точки `z` в скрытом пространстве из нормального распределения, определяемого параметрами `mu` и `log_var`.

❹ Модель Keras, определяющая кодировщик, — модель, которая принимает входное изображение и преобразует его в точку в двумерном скрытом пространстве, выбирая ее из нормального распределения, определяемого параметрами `mu` и `log_var`.

**СЛОЙ LAMBDA**

Слой `Lambda` служит для простого обертывания произвольных функций. Например, следующий слой возводит свои входные значения в квадрат:

```
Lambda (lambda x:x ** 2)
```

Его удобно использовать, когда требуется применить к тензору функцию, еще не включенную в Keras в виде стандартного слоя.

Схема с устройством кодировщика показана на рис. 3.13. Как упоминалось выше, в обычном и вариационном автокодировщике используется один и тот же декодировщик. Единственное, что еще мы должны изменить, — это функция потерь.

## Функция потерь

Прежняя наша функция потерь оценивала только среднеквадратичную ошибку между изображениями и их реконструкциями после прохождения через кодировщик и декодировщик. Эта *потеря при реконструкции* имеется также в вариационном автокодировщике, но нам необходим один дополнительный компонент: *расстояние Кульбака—Лейблера* (Kullback—Leibler divergence). Расстояние Кульбака—Лейблера измеряет, насколько одно распределение вероятности отличается от другого. В вариационном автокодировщике требуется знать, насколько наше нормальное распределение с параметрами `mu` и `log_var` отличается от стандартного нормального распределения. Для данного случая расстояние Кульбака—Лейблера имеет замкнутую форму:

```
kl_loss = -0.5 * sum(1 + log_var – mu ^ 2 – exp(log_var)),
```

или в математической записи:

$$D_{KL}\left[N(\mu,\sigma)\|N(0,1)\right]=\frac{1}{2}\sum\left(1+\log\left(\sigma^2\right)-\mu^2-\sigma^2\right).$$

Сумма вычисляется по всем измерениям в скрытом пространстве. `kl_loss` минимизируется до 0, когда для всех измерений `mu = 0` и `log_var = 0`. Когда эти два члена начинают отличаться от 0, `kl_loss` увеличивается. Таким образом, член с расстоянием Кульбака—Лейблера наказывает сеть за представление наблюдений в виде значений `mu` и `log_var`, значительно отличающихся от параметров стандартного нормального распределения, а именно `mu = 0` и `log_var = 0`. В нашей истории этот член представляет раздражение Эпсилон из-за необходимости отодвигать лестницу от середины стены (`mu` отличается от 0), а также из-за неуверенности мистера Н. Кодера в выборе позиции точки (`log_var` отличается от 0). В обоих случаях растут расходы. Чем же помогает такая добавка в функцию потерь?

Во-первых, имеется четко определенное распределение, пригодное для выбора точек в скрытом пространстве, — стандартное нормальное распределение. Выбирая значение из этого распределения, мы, скорее всего, получим точку, лежащую в пределах того, что видел вариационный автокодировщик (VAE). Во-вторых, поскольку этот член пытается привести все закодированные распределения к стандартному нормальному распределению, уменьшается вероятность возникновения между кластерами точек больших пустых пространств. Вместо этого кодировщик попытается использовать пространство вокруг источника максимально симметрично и эффективно.

В коде функция потерь для VAE выглядит как простое сложение потерь при реконструкции с расстоянием Кульбака—Лейблера. Мы взвешиваем потери при реконструкции значением `r_loss_factor`, гарантирующим согласованность с расстоянием Кульбака—Лейблера. Если придать потерям при реконструкции слишком большой вес, то расстояние Кульбака—Лейблера не будет оказывать желаемого регулирующего эффекта и мы столкнемся с проблемами, присущими обычному автокодировщику. Если же задать слишком маленький вес, то расстояние Кульбака—Лейблера будет доминировать и реконструированные изображения окажутся некачественными. Этот весовой коэффициент — один из параметров, настраиваемых в ходе обучения VAE (листинг 3.8).

**Листинг 3.8.** Включение расстояния Кульбака–Лейблера в функцию потерь

```
### КОМПИЛЯЦИЯ
optimizer = Adam(lr=learning_rate)

def vae_r_loss(y_true, y_pred):
    r_loss = K.mean(K.square(y_true – y_pred), axis = [1,2,3])
    return r_loss_factor * r_loss

def vae_kl_loss(y_true, y_pred):
    kl_loss = -0.5 * K.sum(1 + self.log_var – K.square(self.mu)
                    – K.exp(self.log_var), axis = 1)
    return kl_loss

def vae_loss(y_true, y_pred):
    r_loss = vae_r_loss(y_true, y_pred)
    kl_loss = vae_kl_loss(y_true, y_pred)
    return r_loss + kl_loss

optimizer = Adam(lr=learning_rate)
self.model.compile(optimizer=optimizer, loss = vae_loss
                  , metrics = [vae_r_loss, vae_kl_loss])
```

## Анализ вариационного автокодировщика

Все необходимое для анализа включено в блокнот 03_04_vae_digits_analysis.ipynb, который вы найдете в репозитории с примерами для книги. На рис. 3.10 видно, как изменилась организация скрытого пространства. Черные точки показывают значения mu каждого закодированного изображения. Член, определяющий расстояние Кульбака—Лейблера, гарантирует, что значения mu и sigma не будут сильно отклоняться от стандартного нормального распределения. Соответственно, чтобы сгенерировать новые точки для декодирования (выделены красным), можно сделать выборку из стандартного нормального распределения. Кроме того, у нас получилось не так много сгенерированных цифр, сформированных неправильно, потому что скрытое пространство локально непрерывно из-за того, что кодировщик теперь стохастический, а не детерминированный.

Наконец, окрасив точки в скрытом пространстве разным цветом для разных цифр (рис. 3.14), можно увидеть, что никакие точки не пользуются никакими преференциями. График справа показывает пространство, преобразованное в $p$-значения, и, как видим, все цвета представлены примерно одинаково, причем метки вообще не использовались во время обучения — VAE самостоятельно изучил разные формы цифр, стремясь минимизировать потери при реконструкции.

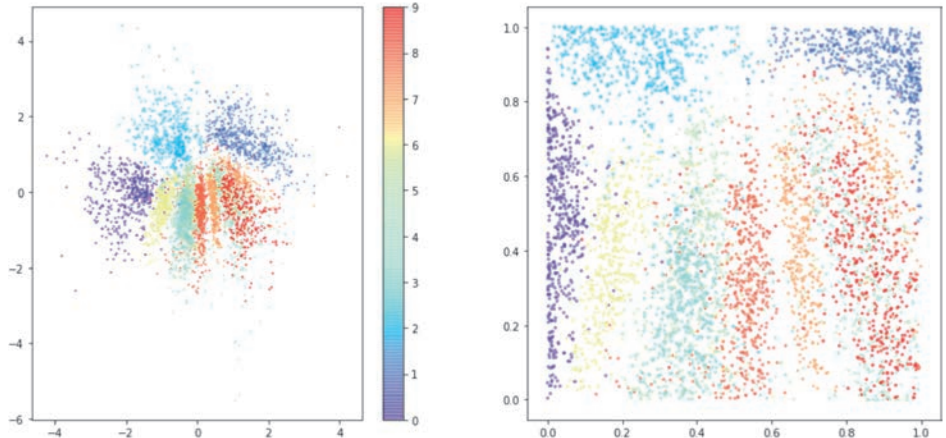



**Рис. 3.14.** Скрытое пространство вариационного автокодировщика, в котором точки, соответствующие разным цифрам, раскрашены в разные цвета

До сих пор, работая над обычным и вариационным автокодировщиком, мы ограничивались двумерным скрытым пространством. Это помогло наглядно показать внутреннюю работу VAE и понять, почему небольшие изменения в архитектуре автокодировщика помогли преобразовать его в более мощный класс сети, пригодной для генеративного моделирования. Теперь обратим внимание на более сложный набор данных и посмотрим, каких удивительных результатов могут достичь вариационные автокодировщики при увеличении размерности скрытого пространства.

## Использование вариационного автокодировщика для генерации изображений лиц

В этом разделе мы используем набор данных CelebFaces Attributes (CelebA, http://bit.ly/2WSiOXt) для обучения нашего следующего вариационного автокодировщика (VAE). Эта коллекция содержит более 200 000 цветных изображений лиц знаменитостей, снабженных различными подписями (например, *в шляпе*, *с улыбкой* и т. д., рис. 3.15). Конечно, нам не нужны метки для обучения VAE, но они пригодятся потом, когда мы начнем исследовать, как эти признаки фиксируются в многомерном скрытом пространстве. После обучения VAE мы сможем выбирать точки из скрытого пространства и генерировать новые изображения лиц знаменитостей.

**Рис. 3.15.** Примеры изображений из набора CelebA[1]

## Обучение VAE

Архитектура сети для моделирования лиц похожа на архитектуру сети из примера с цифрами и имеет лишь несколько небольших отличий:

1. Наши данные теперь имеют три входных канала (RGB) вместо одного (оттенки серого). То есть мы должны изменить количество каналов на 3 в конечном слое обратной свертки декодера.

2. Мы будем использовать скрытое пространство с двумя сотнями измерений вместо двух. Поскольку изображения лиц намного сложнее цифр, мы увеличим размерность скрытого пространства, чтобы сеть смогла закодировать минимально необходимое количество деталей.

3. После каждого сверточного слоя добавлены слои пакетной нормализации для ускорения обучения. Несмотря на то что обработка каждого пакета занимает больше времени, для достижения той же величины потери требуется намного меньше пакетов. Также используются прореживающие слои `Dropout`.

[1] Источник: Лю (Liu) и др., 2015, http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html.

4. Весовой коэффициент потерь при реконструкции увеличен до 10 000. Этот параметр требует настройки; для данного набора данных и данной архитектуры это значение дает хорошие результаты.

5. Для передачи изображений в VAE из папки мы используем *генератор*, чтобы не загружать все изображения в память заранее. Поскольку VAE обучается в пакетном режиме, нет необходимости предварительно загружать все изображения в память, поэтому мы используем встроенный метод `fit_generator`, который Keras предлагает для чтения изображений только тогда, когда они требуются для обучения.

```
Layer (type)                     Output Shape         Param #     Connected to
==========================================================================================
encoder_input (InputLayer)       (None, 128, 128, 3)  0
__________________________________________________________________________________________
encoder_conv_0 (Conv2D)          (None, 64, 64, 32)   896         encoder_input[0][0]
__________________________________________________________________________________________
batch_normalization_1 (BatchNor  (None, 64, 64, 32)   128         encoder_conv_0[0][0]
__________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)        (None, 64, 64, 32)   0           batch_normalization_1[0][0]
__________________________________________________________________________________________
dropout_1 (Dropout)              (None, 64, 64, 32)   0           leaky_re_lu_1[0][0]
__________________________________________________________________________________________
encoder_conv_1 (Conv2D)          (None, 32, 32, 64)   18496       dropout_1[0][0]
__________________________________________________________________________________________
batch_normalization_2 (BatchNor  (None, 32, 32, 64)   256         encoder_conv_1[0][0]
__________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)        (None, 32, 32, 64)   0           batch_normalization_2[0][0]
__________________________________________________________________________________________
dropout_2 (Dropout)              (None, 32, 32, 64)   0           leaky_re_lu_2[0][0]
__________________________________________________________________________________________
encoder_conv_2 (Conv2D)          (None, 16, 16, 64)   36928       dropout_2[0][0]
__________________________________________________________________________________________
batch_normalization_3 (BatchNor  (None, 16, 16, 64)   256         encoder_conv_2[0][0]
__________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)        (None, 16, 16, 64)   0           batch_normalization_3[0][0]
__________________________________________________________________________________________
dropout_3 (Dropout)              (None, 16, 16, 64)   0           leaky_re_lu_3[0][0]
__________________________________________________________________________________________
encoder_conv_3 (Conv2D)          (None, 8, 8, 64)     36928       dropout_3[0][0]
__________________________________________________________________________________________
batch_normalization_4 (BatchNor  (None, 8, 8, 64)     256         encoder_conv_3[0][0]
__________________________________________________________________________________________
leaky_re_lu_4 (LeakyReLU)        (None, 8, 8, 64)     0           batch_normalization_4[0][0]
__________________________________________________________________________________________
dropout_4 (Dropout)              (None, 8, 8, 64)     0           leaky_re_lu_4[0][0]
__________________________________________________________________________________________
flatten_1 (Flatten)              (None, 4096)         0           dropout_4[0][0]
__________________________________________________________________________________________
mu (Dense)                       (None, 200)          819400      flatten_1[0][0]
__________________________________________________________________________________________
log_var (Dense)                  (None, 200)          819400      flatten_1[0][0]
__________________________________________________________________________________________
encoder_output (Lambda)          (None, 200)          0           mu[0][0]
                                                                  log_var[0][0]
==========================================================================================
Total params: 1,732,944
Trainable params: 1,732,496
Non-trainable params: 448
__________________________________________________________________________________________
```

**Рис. 3.16.** Кодировщик VAE для обработки набора данных CelebA

На рис. 3.16 и 3.17 показана полная архитектура кодировщика и декодировщика.

Чтобы обучить VAE на наборе данных CelebA, запустите блокнот Jupyter Notebook 03_05_vae_faces_train.ipynb из репозитория с примерами для книги. После пяти эпох обучения VAE должен быть готов генерировать новые изображения лиц знаменитостей!

```
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   (None, 200)               0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              823296
_________________________________________________________________
reshape_1 (Reshape)          (None, 8, 8, 64)          0
_________________________________________________________________
decoder_conv_t_0 (Conv2DTran (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
dropout_5 (Dropout)          (None, 16, 16, 64)        0
_________________________________________________________________
decoder_conv_t_1 (Conv2DTran (None, 32, 32, 64)        36928
_________________________________________________________________
batch_normalization_6 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 32, 32, 64)        0
_________________________________________________________________
dropout_6 (Dropout)          (None, 32, 32, 64)        0
_________________________________________________________________
decoder_conv_t_2 (Conv2DTran (None, 64, 64, 32)        18464
_________________________________________________________________
batch_normalization_7 (Batch (None, 64, 64, 32)        128
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 64, 64, 32)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 64, 64, 32)        0
_________________________________________________________________
decoder_conv_t_3 (Conv2DTran (None, 128, 128, 3)       867
_________________________________________________________________
activation_1 (Activation)    (None, 128, 128, 3)       0
=================================================================
Total params: 917,123
Trainable params: 916,803
Non-trainable params: 320
_________________________________________________________________
```

**Рис. 3.17.** Декодировщик VAE для обработки набора данных CelebA

## Анализ вариационного автокодировщика

Повторить анализ, описываемый ниже, можно, запустив блокнот 03_06_vae_faces_analysis.ipynb сразу после обучения VAE. Многие идеи, представленные в этом разделе, были почерпнуты из следующей статьи.[1] Для начала рассмотрим примеры реконструированных лиц. В верхнем ряду на рис. 3.18 показаны исходные изображения, а в нижнем ряду — их реконструкции, полученные после прохождения через кодировщик и декодировщик.

VAE благополучно уловил ключевые черты каждого лица — угол наклона головы, прическу, выражение лица и т. д. Некоторые мелкие детали отсутствуют, но не забывайте, что цель создания вариационного автокодировщика состоит не в том, чтобы добиться идеальной потери при реконструкции. Наша конечная цель — создание новых лиц путем выборки случайных точек из скрытого пространства. Для этого требуется гарантировать близость распределения точек в скрытом пространстве к многомерному стандартному нормальному распределению. Мы не можем просматривать все измерения одновременно, поэтому будем проверять распределение каждого скрытого измерения в отдельности. Если обнаружатся какие-либо измерения, значительно отличающиеся от стандартного нормального распределения, то нам, вероятно, следует уменьшить весовой коэффициент потерь при реконструкции, поскольку член, определяющий расстояние Кульбака—Лейблера, не оказывает достаточного влияния. На рис. 3.19 показаны первые 50 измерений в нашем скрытом пространстве. Как видим, все распределения мало отличаются от стандартного нормального, поэтому мы можем перейти к созданию некоторых лиц!

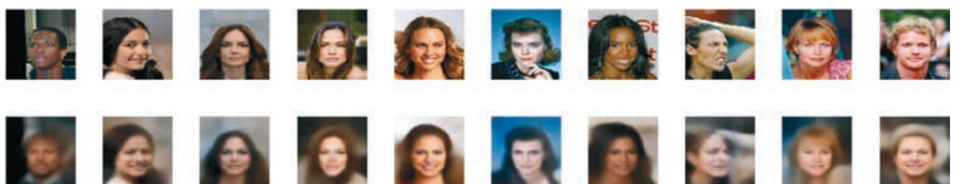


**Рис. 3.18.** Реконструированные лица, полученные после прохождения через кодировщик и декодировщик

[1] Сянсу Хоу (Xianxu Hou) и др., «Deep Feature Consistent Variational Autoencoder», 2 октября 2016, https://arxiv.org/abs/1610.00291.

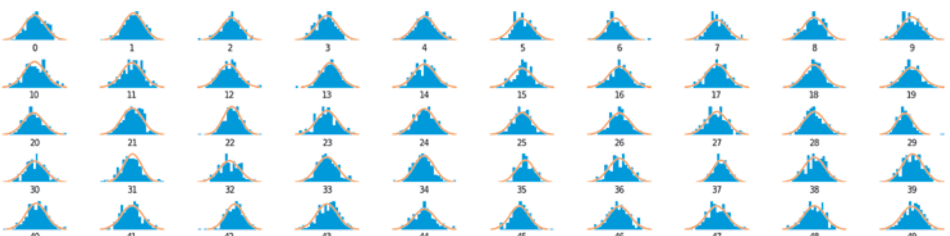


**Рис. 3.19.** Распределение точек для первых 50 измерений скрытого пространства

## Генерирование новых лиц

Сгенерировать новые лица можно с помощью кода, представленного в листинге 3.9.

**Листинг 3.9.** Генерирование новых лиц по точкам в скрытом пространстве

```
n_to_show = 30

znew = np.random.normal(size = (n_to_show,VAE.z_dim)) ❶

reconst = VAE.decoder.predict(np.array(znew)) ❷

fig = plt.figure(figsize=(18, 5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(n_to_show):
    ax = fig.add_subplot(3, 10, i+1)
    ax.imshow(reconst[i, :,:,:]) ❸
    ax.axis('off')

plt.show()
```

❶ Выбираем 30 точек из 200-мерного стандартного нормального распределения...

❷ ...и передаем их в декодировщик.

❸ В результате получаем изображения 128 × 128 × 3, которые можно просмотреть (рис. 3.20).

Удивительно, но VAE смог преобразовать каждую выбранную нами точку в довольно реалистичное изображение лица человека. Да, изображения не идеальны, но они разительно отличаются в лучшую сторону от тех, что были

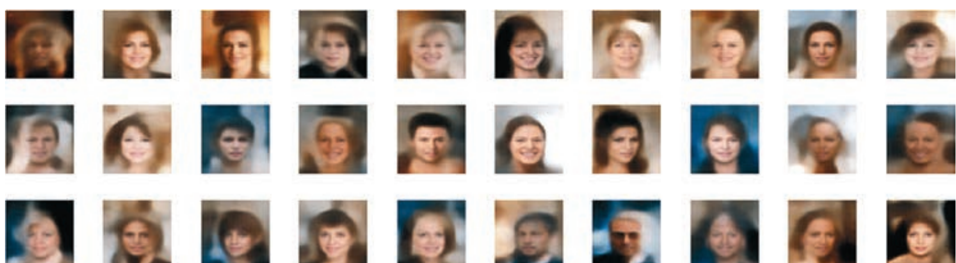


**Рис. 3.20.** Новые сгенерированные лица

получены с наивной байесовской моделью (см. главу 1), которая не учитывала зависимости между соседними пикселами, поскольку не поддерживала понятия *высокоуровневых признаков*, таких как *солнцезащитные очки* или *каштановые волосы*. VAE не имеет этой проблемы, поскольку сверточные слои кодировщика способны преобразовать низкоуровневые пикселы в высокоуровневые признаки, а декодировщик обучен решать противоположную задачу преобразования высокоуровневых признаков из скрытого пространства обратно в простые пикселы.

## Арифметика скрытого пространства

Одним из преимуществ отображения изображений в пространство меньшего размера является возможность выполнения арифметических операций с векторами в скрытом пространстве, результаты которых можно преобразовать в визуальные аналоги путем декодирования в пространство исходных изображений. Например, предположим, что мы решили преобразовать изображение грустного лица в улыбающееся. Для этого сначала нужно найти вектор в скрытом пространстве, который указывает в направлении увеличения признака «улыбка». Прибавив этот вектор к кодированному исходному изображению в скрытом пространстве, мы получим новую точку, которая при декодировании должна дать улыбающуюся версию исходного изображения.

Но как найти вектор, соответствующий признаку «улыбка»? Каждое изображение в наборе данных CelebA отмечено атрибутами, один из которых — *с улыбкой*. Если взять среднюю позицию закодированных изображений в скрытом пространстве с атрибутом *с улыбкой* и вычесть среднюю позицию закодированных изображений без этого атрибута, то мы получим вектор, который указывает в направлении увеличения признака «улыбка». Факти-

чески мы выполним следующую арифметическую операцию с векторами в скрытом пространстве, где `alpha` — коэффициент, определяющий степень увеличения или уменьшения признака, соответствующего вектору:

```
z_new = z + alpha * feature_vector
```

Посмотрим, как это работает. На рис. 3.21 показано несколько изображений, закодированных в скрытое пространство. Мы добавляем или вычитаем вектор (например улыбка, блондинка, мужчина, очки), умноженный на коэффициент, чтобы получить разные версии изображения, в которых меняется только требуемый признак.

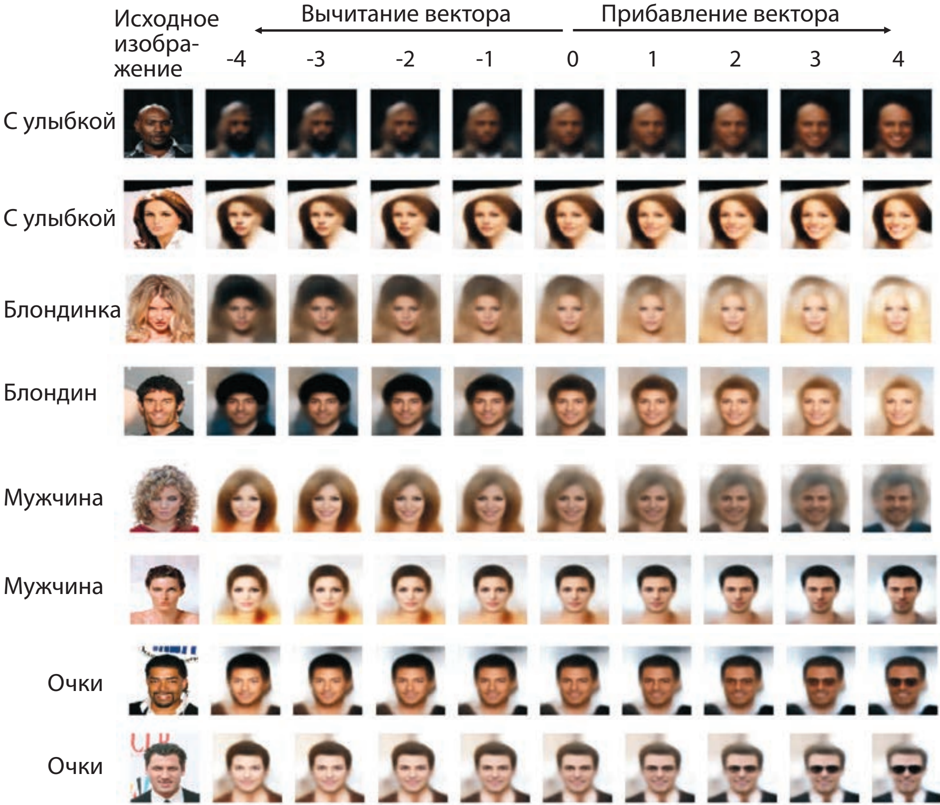



**Рис. 3.21.** Добавление и вычитание признаков из изображений лиц

Весьма примечательно, что даже при значительном перемещении точки в скрытом пространстве основная часть изображения практически не меняется, кроме признака, которым мы пытаемся манипулировать. Это демонстрирует мощную способность вариационных автокодировщиков выявлять и корректировать высокоуровневые признаки в изображениях.

## Преобразование одного лица в другое

Эту же идею можно использовать и для преобразования одного лица в другое. Представьте две точки в скрытом пространстве, A и B, соответствующие двум изображениям. Если пройти из точки A в точку B по прямой, декодируя каждую точку линии по мере движения, то можно увидеть постепенное превращение начального лица в конечное. Математически эту прямую линию можно описать следующим уравнением:

```
z_new = z_A * (1- alpha) + z_B * alpha
```

Здесь `alpha` — это число между 0 и 1, определяющее расстояние от начальной точки A (рис. 3.22). Мы берем два изображения, кодируем их в скрытое пространство, а затем декодируем точки, лежащие на прямой линии между ними через равные промежутки.

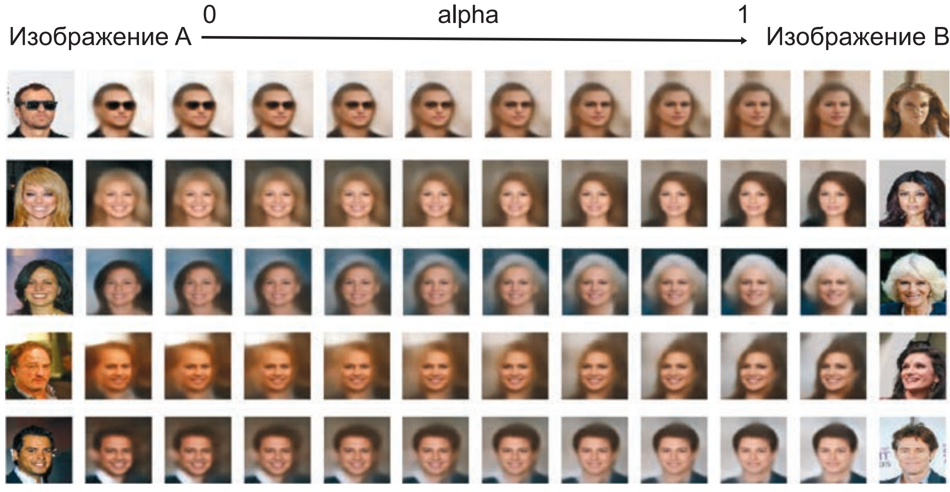



**Рис. 3.22.** Преобразование одного лица в другое

Обратите внимание, насколько плавным выглядит переход — даже при изменении нескольких признаков одновременно: удалении очков, изменении цвета волос и пола и др. Такая плавность еще раз доказывает, что скрытое пространство вариационного автокодировщика действительно непрерывно, его можно обойти и исследовать, создав множество разных человеческих лиц.

## Итоги

В этой главе мы узнали, что вариационные автокодировщики являются мощным инструментом в арсенале генеративного моделирования. Сначала мы посмотрели, как можно использовать простые автокодировщики для отображения многомерных изображений в скрытое пространство с небольшим числом измерений, чтобы из отдельных неинформативных пикселов извлечь высокоуровневые признаки. Однако, как и в случае с художественной выставкой братьев Кодер, мы быстро выяснили, что простые автокодировщики из-за своих недостатков малопригодны на роль генеративной модели — выбор представительных точек из полученного скрытого пространства сопряжен с некоторыми проблемами по ряду причин. Вариационные автокодировщики решают эти проблемы, вводя в модель элемент случайности и ограничивая распределение точек в скрытом пространстве. С небольшими изменениями обычный автокодировщик можно преобразовать в вариационный, что позволяет использовать его в качестве генеративной модели.

Наконец, мы применили новую технику для генерации лиц и увидели, что для получения новых лиц можно просто выбирать точки из стандартного нормального распределения. Более того, выполняя арифметические операции с векторами в скрытом пространстве, можно достичь некоторых удивительных эффектов, таких как трансформация лица и манипулирование объектами. Эти эксперименты наглядно показали, почему вариационные автокодировщики приобрели популярность в задачах генеративного моделирования в последние годы.

В следующей главе мы рассмотрим тип генеративной модели, привлекающей еще большее внимание: генеративно-состязательную сеть.

ГЛАВА 4

# Генеративно-состязательные сети

Пятого декабря 2016 года на конференции Neural Information Processing Systems (NIPS) в Барселоне Ян Гудфеллоу из Google Brain представил учебное руководство под названием «Generative Adversarial Networks» (https://bit.ly/2WkkB7r).[1] Идеи, представленные в руководстве, теперь рассматриваются как один из поворотных моментов в развитии генеративного моделирования, и с тех пор появилось множество их вариаций, которые подняли эту область на еще большую высоту. В этой главе мы сначала познакомимся с теоретической основой генеративно-состязательных сетей (Generative Adversarial Networks, GAN). Затем вы узнаете, как с помощью библиотеки Keras для Python создавать свои генеративно-состязательные сети. Но сначала мы отправимся в джунгли, чтобы встретиться с Джином.

## Ганимал

Однажды, прогуливаясь по местным джунглям, Джин увидел женщину, листающую черно-белые фотографии и явно чем-то обеспокоенную. Он подошел и спросил, может ли чем-то помочь ей. Женщина представилась как Ди, пояснив, что охотится за неуловимым *ганималом*, мифическим существом, которое, как говорят, бродит по джунглям. Поскольку существо ведет ночной образ жизни, у Ди есть только коллекция ночных фотографий ганимала, которые оставили в джунглях другие исследователи (рис. 4.1).

[1] Ian Goodfellow, «NIPS 2016 Tutorial: Generative Adversarial Networks», 21 декабря 2016, https://arxiv.org/abs/1701.00160v4.

Ди зарабатывает, продавая фотографии коллекционерам, и обеспокоена тем, что никогда не видела ганимала и что ее бизнес угаснет, если она не сможет в ближайшее время сделать новые фотографии.

Будучи заядлым фотографом, Джин решает помочь Ди. Он соглашается сделать для нее новые фотографии ночного зверя. Однако Джин никогда не видел ганимала и не знает, как сделать хорошие фотографии. Кроме того, поскольку Ди продавала только найденные ею фотографии, она не сможет отличить хорошую фотографию ганимала от фотографии среднего качества. Учитывая полное отсутствие информации о ганимале, как добиться, чтобы Джин в конечном итоге сделал впечатляющие фотографии?

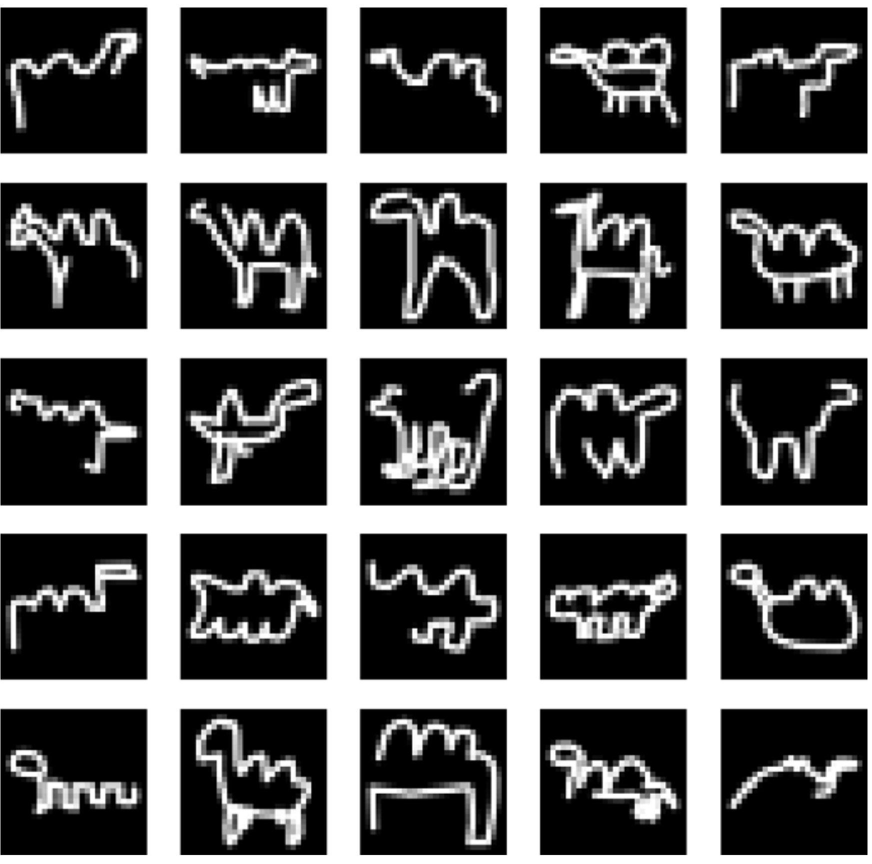


**Рис. 4.1.** Оригинальные фотографии ганимала

Ди и Джин придумали следующее. Каждую ночь Джин делает 64 фотографии в разных местах с разными случайными выдержками и смешивает их с 64 фотографиями ганимала из оригинальной коллекции. Затем Ди просматривает этот набор фотографий и пытается угадать, какие были сделаны Джином, а какие являются оригиналами. Основываясь на своих ошибках, она учится отличать фотографии Джина от оригинальных. После этого Джин делает еще 64 фотографии и вновь показывает их Ди. Ди присваивает каждой фотографии оценку от 0 до 1, в зависимости от степени кажущейся реалистичности. Основываясь на этих оценках, Джин перенастраивает свою камеру, чтобы в следующий раз сделать фотографии, которым Ди присвоит высокую оценку с большей вероятностью, и т. д.

Этот процесс продолжался много дней и недель. Первое время Джин не получал никакой полезной обратной связи от Ди, так как она лишь случайно угадывала подлинные фотографии. Однако спустя несколько недель обучения она стала различать их лучше и, соответственно, смогла обеспечить более ценную обратную связь, на основе которой Джин мог настроить свою камеру. Это усложнило задачу для Ди, так как теперь фотографии Джина стало сложнее отличить от настоящих, поэтому ей пришлось снова совершенствовать свои умения. Со временем Джин все лучше и лучше справлялся с производством фотографий ганимала, и в конце концов Ди отметила, что не может отличить фотографии Джина от оригиналов. Они взяли фотографии, сделанные Джином, выставили их на аукцион, и эксперты были поражены качеством новых наблюдений — они были так же убедительны, как и оригиналы (рис. 4.2).

**Рис. 4.2.** Примеры фотографий ганимала, сделанные Джином

## Введение в генеративно-состязательные сети

Приключения Джина и Ди, охотящихся на неуловимого ночного ганимала, — это метафора одного из самых важных достижений глубокого обучения последних лет: создания генеративно-состязательных сетей. Проще говоря, генеративно-состязательная сеть — это битва между двумя противоборствующими сторонами: генератором и дискриминатором. Генератор пытается преобразовать случайный шум в наблюдения, которые выглядят так, как будто были выбраны из исходного набора данных, а дискриминатор старается определить, исходит ли наблюдение из оригинального набора данных или является одной из подделок генератора (рис. 4.3).

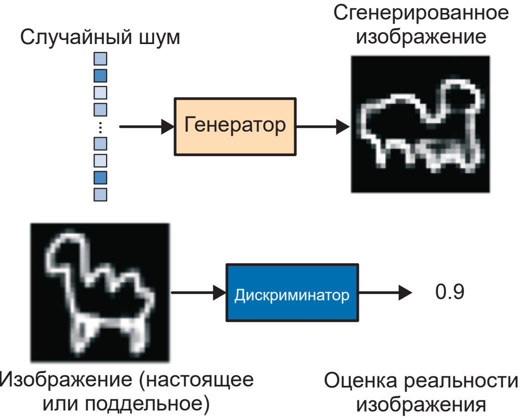



**Рис. 4.3.** Входы и выходы двух подсетей в генеративно-состязательной сети

В начале процесса генератор выводит искаженные изображения, а дискриминатор оценивает их случайным образом. Главная особенность генеративно-состязательной сети — попеременное обучение двух подсетей: генератор постепенно совершенствуется в обмане дискриминатора, а дискриминатор, адаптируясь, учится правильно определять фальшивые наблюдения, что заставлет генератор искать новые способы обмана дискриминатора, и т. д. Чтобы увидеть процесс в действии, построим генеративно-состязательную сеть, формирующую фотографии ночных ганималов.

## Ваша первая генеративно-состязательная сеть

Прежде всего вам нужно загрузить обучающие данные. Мы будем использовать набор данных Quick, Draw! (http://bit.ly/30HyNqg) от компании Google. Это общедоступная коллекция черно-белых рисунков размером 28 × 28 пикселов, маркированных по темам. Набор данных был собран в ходе онлайн-игры, в которой игрокам предлагалось нарисовать изображение, представляющее объект или идею, а нейронная сеть пыталась угадать этот объект. Это действительно полезный и забавный набор данных, который можно использовать для знакомства с основами глубокого обучения. Чтобы получить этот набор данных, нужно загрузить camel в формате `numpy` и сохранить его в папке ./data/camel/, внутри репозитория с примерами для книги.[1] Исходные данные содержат значения в диапазоне [0, 255], обозначающие интенсивность окраски пикселов. Для нужд нашей сети мы преобразуем их в диапазон [–1, 1]. Запустив блокнот 04_01_gan_camel_train.ipynb из репозитория с примерами, начнем обучение генеративно-состязательной сети. Как и в предыдущей главе, рассказывающей о вариационных автокодировщиках, вы можете создать экземпляр сети в блокноте (листинг 4.1) и поэкспериментировать с параметрами, чтобы увидеть, как они влияют на модель.

**Листинг 4.1.** Определение генеративно-состязательной сети

```
gan = GAN(input_dim = (28,28,1)
            , discriminator_conv_filters = [64,64,128,128]
            , discriminator_conv_kernel_size = [5,5,5,5]
            , discriminator_conv_strides = [2,2,2,1]
            , discriminator_batch_norm_momentum = None
            , discriminator_activation = 'relu'
            , discriminator_dropout_rate = 0.4
            , discriminator_learning_rate = 0.0008
            , generator_initial_dense_layer_size = (7, 7, 64)
            , generator_upsample = [2,2, 1, 1]
            , generator_conv_filters = [128,64, 64,1]
            , generator_conv_kernel_size = [5,5,5,5]
            , generator_conv_strides = [1,1, 1, 1]
            , generator_batch_norm_momentum = 0.9
            , generator_activation = 'relu'
            , generator_dropout_rate = None
            , generator_learning_rate = 0.0004
```

[1] По счастливому совпадению ганималы очень похожи на верблюдов. (Имя файла camel переводится как «верблюд», то есть он содержит рисунки, изображающие верблюдов. — *Примеч. пер.*)

```
            , optimiser = 'rmsprop'
            , z_dim = 100
            )
```

Сначала выясним, как выглядит конструкция дискриминатора.

## Дискриминатор

Цель дискриминатора — определить реальность изображения. Это — задача классификации изображений с учителем, поэтому мы можем использовать ту же сетевую архитектуру, что и в главе 2: последовательность сверточных слоев, за которыми следует выходной полносвязанный слой.

В оригинальной статье, описывающей работу генеративно-состязательной сети (GAN), вместо сверточных использовались полносвязанные слои. Однако впоследствии было не раз показано, что дискриминатор со сверточными слоями обладает большей предсказательной силой. В литературе сети этого типа иногда называют глубокими сверточными генеративно-состязательными сетями (Deep Convolutional Generative Adversarial Network, DCGAN), но в настоящее время почти все архитектуры GAN содержат сверточные слои, поэтому, когда мы говорим о GAN, приставка «DC» воспринимается как само собой разумеющаяся. Также в дискриминаторах оригинальных GAN часто встречаются слои пакетной нормализации, но для простоты мы не будем их использовать. На рис. 4.4 показана полная архитектура дискриминатора, которую мы построим.

В листинге 4.2 представлен код, использующий Keras и конструирующий дискриминатор.

**Листинг 4.2.** Дискриминатор

```
discriminator_input = Input(shape=self.input_dim,
                            name='discriminator_input') ❶
x = discriminator_input

for i in range(self.n_layers_discriminator): ❷

    x = Conv2D(
        filters = self.discriminator_conv_filters[i]
        , kernel_size = self.discriminator_conv_kernel_size[i]
```

```
        , strides = self.discriminator_conv_strides[i]
        , padding = 'same'
        , name = 'discriminator_conv_' + str(i)
        )(x)

    if self.discriminator_batch_norm_momentum and i > 0:
        x = BatchNormalization(momentum = self.discriminator_batch_norm_
                               momentum)(x)

    x = Activation(self.discriminator_activation)(x)

    if self.discriminator_dropout_rate:
        x = Dropout(rate = self.discriminator_dropout_rate)(x)

x = Flatten()(x) ❸
discriminator_output= Dense(1, activation='sigmoid'
    , kernel_initializer = self.weight_init)(x) ❹

discriminator = Model(discriminator_input, discriminator_output) ❺
```

❶ Определение входного слоя дискриминатора (для приема изображений).

❷ Набор сверточных слоев, следующих друг за другом.

❸ Преобразование выхода последнего сверточного слоя в вектор.

❹ Слой `Dense`, содержащий единственный узел с функцией активации *sigmoid*, которая преобразует результат полносвязанного слоя в диапазон [0, 1].

❺ Модель Keras, определяющая дискриминатор, — модель, которая принимает исходное изображение и выводит единственное число между 0 и 1.

Отметим, что в некоторых сверточных слоях используется шаг с величиной 2, позволяющий уменьшить размер тензора при прохождении через сеть, при этом мы увеличиваем количество каналов (от 1 в исходном черно-белом изображении до 64 и затем до 128). Функция активации *sigmoid* в заключительном слое обеспечивает масштабирование выходного значения в диапазон от 0 до 1. Оно будет играть роль оценки достоверности изображения.

```
Layout (type)                    Output Shape              Param #
=================================================================
discriminator_input (InputLa (None, 28, 28, 1)         0
_________________________________________________________________
discriminator_conv_0 (Conv2D (None, 14, 14, 64)        1664
_________________________________________________________________
activation_1 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
discriminator_conv_1 (Conv2D (None, 7, 7, 64)          102464
_________________________________________________________________
activation_2 (Activation)    (None, 7, 7, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
discriminator_conv_2 (Conv2D (None, 4, 4, 128)         204928
_________________________________________________________________
activation_3 (Activation)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
discriminator_conv_3 (Conv2D (None, 4, 4, 128)         409728
_________________________________________________________________
activation_4 (Activation)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_4 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 2049
=================================================================
Total params: 720,833
Trainable params: 720,833
Non-trainable params: 0
_________________________________________________________________
```

**Рис. 4.4.** Дискриминатор генеративно-состязательной сети

## Генератор

Теперь сконструируем генератор. На вход генератора подается вектор, обычно получаемый из многомерного стандартного нормального распределения. На выходе получается изображение того же размера, что и изображения в исходных обучающих данных. Конструкцией генератор напоминает декодер в вариационном автокодировщике. Фактически генератор в генеративно-состязательной сети решает ту же задачу, что и декодер в вариационном автокодировщике, преобразуя вектор из скрытого пространства в изо-

бражение. Идея преобразования точки из скрытого пространства обратно в исходную область очень распространена в генеративном моделировании, поскольку дает возможность манипулировать векторами в скрытом пространстве и тем самым изменять высокоуровневые характеристики изображений в исходной области (рис. 4.5).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
generator_input (InputLayer) (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 3136)              316736
_________________________________________________________________
batch_normalization_10 (Batc (None, 3136)              12544
_________________________________________________________________
activation_36 (Activation)   (None, 3136)              0
_________________________________________________________________
reshape_4 (Reshape)          (None, 7, 7, 64)          0
_________________________________________________________________
up_sampling2d_10 (UpSampling (None, 14, 14, 64)        0
_________________________________________________________________
generator_conv_0 (Conv2D)    (None, 14, 14, 128)       204928
_________________________________________________________________
batch_normalization_11 (Batc (None, 14, 14, 128)       512
_________________________________________________________________
activation_37 (Activation)   (None, 14, 14, 128)       0
_________________________________________________________________
up_sampling2d_11 (UpSampling (None, 28, 28, 128)       0
_________________________________________________________________
generator_conv_1 (Conv2D)    (None, 28, 28, 64)        204864
_________________________________________________________________
batch_normalization_12 (Batc (None, 28, 28, 64)        256
_________________________________________________________________
activation_38 (Activation)   (None, 28, 28, 64)        0
_________________________________________________________________
generator_conv_2 (Conv2D)    (None, 28, 28, 64)        102464
_________________________________________________________________
batch_normalization_13 (Batc (None, 28, 28, 64)        256
_________________________________________________________________
activation_39 (Activation)   (None, 28, 28, 64)        0
_________________________________________________________________
generator_conv_3 (Conv2D)    (None, 28, 28, 1)         1601
_________________________________________________________________
activation_40 (Activation)   (None, 28, 28, 1)         0
=================================================================
Total params: 844,161
Trainable params: 837,377
Non-trainable params: 6,784
```

**Рис. 4.5.** Генератор

### УВЕЛИЧЕНИЕ РАЗРЕШЕНИЯ

В декодере вариационного автокодировщика, созданного в предыдущей главе, мы удваивали ширину и высоту тензора с помощью слоев `Conv2DTranspose` с величиной шага 2. Эти слои вставляли нулевые значения между пикселами перед выполнением операции свертки.

В этой сети для удвоения ширины и высоты входного тензора мы используем слой `Upsampling2D`, который просто дублирует каждую строку и столбец во входных данных. За ним следует обычный сверточный слой с размером шага 1, выполняющий операцию свертки. Идея — та же, что и в обратной свертке, но вместо заполнения промежутков между пикселами нулями операция увеличения разрешения просто повторяет значения существующих пикселов.

Оба метода — `Upsampling2D + Conv2D` и `Conv2DTranspose` — с успехом можно использовать для преобразования точки из скрытого пространства обратно в область исходных изображений. Это как раз тот случай, когда желательно попробовать оба метода и посмотреть, какой из них позволяет получить лучшие результаты. По опыту, метод

**Рис. 4.6.** Артефакты, появляющиеся при использовании слоев Conv2DTranspose[1]

[1] Источник: Огастес Одена (Augustus Odena) и др., «Deconvolution and Checkerboard Artifacts», 17 октября 2016, http://bit.ly/31MgHUQ.

`Conv2DTranspose` может привести к появлению *артефактов,* или эффекта шахматной доски, в выходном изображении (рис. 4.6), снижающих качество результата. Тем не менее они все еще используются во многих весьма впечатляющих генеративно-состязательных сетях в литературе и доказали, что являются мощным инструментом в арсенале практиков глубокого обучения. И снова я советую поэкспериментировать с обоими методами и посмотреть, какой из них лучше подходит для вашей задачи.

Теперь познакомимся с новым типом слоев: слоем *увеличения разрешения* (upsampling layer).

В листинге 4.3 представлен код, конструирующий генератор.

**Листинг 4.3.** Генератор

```
generator_input = Input(shape=(self.z_dim,), name='generator_input') ❶
x = generator_input

x = Dense(np.prod(self.generator_initial_dense_layer_size))(x) ❷

if self.generator_batch_norm_momentum:
    x = BatchNormalization(momentum = self.generator_batch_norm_momentum)(x)

x = Activation(self.generator_activation)(x)

x = Reshape(self.generator_initial_dense_layer_size)(x) ❸

if self.generator_dropout_rate:
    x = Dropout(rate = self.generator_dropout_rate)(x)

for i in range(self.n_layers_generator): ❹

    x = UpSampling2D()(x)
    x = Conv2D(
        filters = self.generator_conv_filters[i]
        , kernel_size = self.generator_conv_kernel_size[i]
        , padding = 'same'
        , name = 'generator_conv_' + str(i)
        )(x)

    if i < n_layers_generator - 1: ❺
        if self.generator_batch_norm_momentum:
            x = BatchNormalization(momentum = self.generator_batch_norm_
                                   momentum))(x)
        x = Activation('relu')(x)
```

```
    else:
        x = Activation('tanh')(x)

generator_output = x
generator = Model(generator_input, generator_output) ❻
```

❶ Определение входного слоя генератора — вектор с длиной 100.

❷ За ним следует слой `Dense` с 3136 узлами...

❸ ...который, после применения пакетной нормализации и функции активации ReLU, преобразуется в тензор с формой 7 × 7 × 64.

❹ Далее тензор пропускается через четыре слоя `Conv2D`, первым двум из которых предшествуют слои `Upsampling2D`, увеличивающие размер тензора сначала до 14 × 14, а затем до 28 × 28 (размер исходных изображений). Во всех слоях, кроме последнего, используется пакетная нормализация и функция активации ReLU (также можно было бы использовать LeakyReLU).

❺ После последнего слоя `Conv2D` используется активация *tanh*, чтобы преобразовать выходные значения в диапазон [–1, 1], в соответствии с областью исходных изображений.

❻ Модель Keras, определяющая генератор, — модель, которая принимает вектор с длиной 100 и возвращает тензор с формой [28, 28, 1].

## Обучение генеративно-состязательной сети

Генератор и дискриминатор в GAN имеют очень простую архитектуру, не сильно отличающуюся от моделей, которые мы рассматривали ранее. Ключом к пониманию GAN является понимание процесса обучения. Для обучения дискриминатора нужно создать обучающий набор, в котором часть изображений является случайно выбранными *реальными* наблюдениями, а часть — результатом работы генератора. Для истинных изображений ответ должен быть равным 1, а для сгенерированных — 0. Если рассматривать эту задачу как обучение с учителем, то мы должны обучить дискриминатор отличать исходные и сгенерированные изображения и выводить значения, близкие к 1, для истинных изображений и значения, близкие к 0, — для поддельных.

Обучение генератора выглядит сложнее, так как в этом случае нет обучающего набора, с помощью которого можно было бы определить *истинное*

изображение, которому должна соответствовать конкретная точка в скрытом пространстве. Вместо этого генератор должен научиться генерировать такие изображения, которые дискриминатор не сможет отличить от истинных, то есть когда такое изображение будет подано на вход дискриминатора, тот должен вернуть значение, близкое к 1. Чтобы обучить генератор, мы должны подключить его к дискриминатору и создать модель Keras, которую затем можно будет обучить. В частности, мы должны передать выход генератора (изображение 28 × 28 × 1) на вход дискриминатора и на выходе этой комбинированной модели обеспечить вероятность того, что сгенерированное изображение является *реальным* с точки зрения дискриминатора. Для обучения комбинированной модели создадим обучающие пакеты, состоящие из случайно сгенерированных 100-мерных скрытых векторов с ответом, равным 1, поскольку мы хотим обучить генератор создавать изображения, которые дискриминатор считает реальными.

В данном случае функция потерь — это просто бинарная перекрестная энтропия между выходом дискриминатора и вектором ответа 1. Подчеркнем: мы должны зафиксировать веса дискриминатора на время обучения комбинированной модели, чтобы обновлялись только веса генератора. В противном случае дискриминатор будет корректировать веса, стремясь с большей вероятностью определить сгенерированные изображения как реальные, что нежелательно. Нам нужно, чтобы сгенерированные изображения определялись как близкие к реальным именно вследствие преимуществ генератора, а не потому, что дискриминатору присущи недостатки (рис. 4.7).

Теперь посмотрим, как это выглядит в коде. Сначала нужно скомпилировать модель дискриминатора и модель, которая обучает генератор (листинг 4.4).

**Листинг 4.4.** Компиляция генеративно-состязательной сети

```
### КОМПИЛЯЦИЯ МОДЕЛИ, ОБУЧАЮЩЕЙ ДИСКРИМИНАТОР

self.discriminator.compile(
    optimizer= RMSprop(lr=0.0008)
    , loss = 'binary_crossentropy'
    , metrics = ['accuracy']
) ❶

### КОМПИЛЯЦИЯ МОДЕЛИ, ОБУЧАЮЩЕЙ ГЕНЕРАТОР

self.discriminator.trainable = False ❷
model_input = Input(shape=(self.z_dim,), name='model_input')
model_output = discriminator(self.generator(model_input))
```

```
self.model = Model(model_input, model_output) ❸

self.model.compile(
    optimizer=RMSprop(lr=0.0004)
    , loss='binary_crossentropy'
    , metrics=['accuracy']
) ❹
```

❶ Дискриминатор компилируется с функцией потерь бинарной перекрестной энтропии, поскольку ответ имеет бинарный характер, и на выходе есть единственный узел с функцией активации *sigmoid*.

❷ Затем мы фиксируем веса дискриминатора — это не влияет на уже скомпилированную модель дискриминатора.

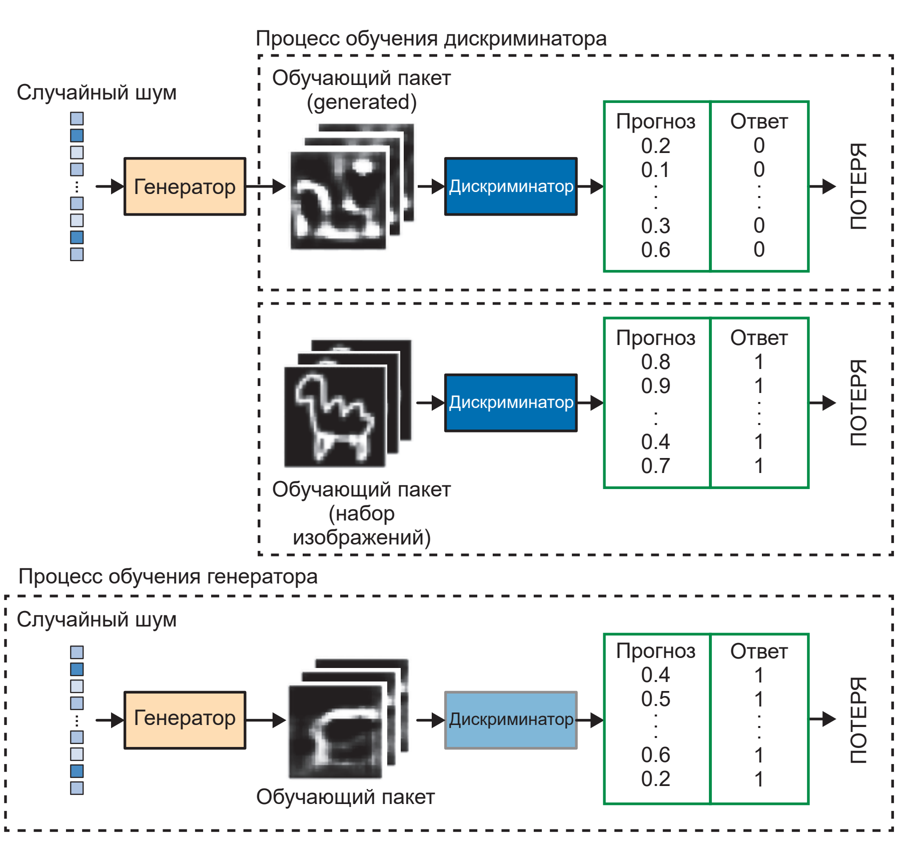



**Рис. 4.7.** Обучение генеративно-состязательной сети

❸ Определяем новую модель, на вход которой подается 100-мерный скрытый вектор; он передается через генератор и зафиксированный дискриминатор с целью получения вероятности на выходе.

❹ Для комбинированной модели повторно используем бинарную перекрестную энтропию в роли функции потерь — обучаться такая модель будет медленнее, чем дискриминатор, потому что часто желательно, чтобы дискриминатор был сильнее генератора. Скорость обучения — это параметр, к настройке которого следует подходить с особой осторожностью, независимо от задачи, которую призвана решать генеративно-состязательная сеть.

Обучим сеть, чередуя обучение дискриминатора и генератора (листинг 4.5).

**Листинг 4.5.** Обучение сети

```
def train_discriminator(x_train, batch_size):

    valid = np.ones((batch_size,1))
    fake = np.zeros((batch_size,1))

    # ОБУЧЕНИЕ НА РЕАЛЬНЫХ ИЗОБРАЖЕНИЯХ
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    true_imgs = x_train[idx]
    self.discriminator.train_on_batch(true_imgs, valid) ❶

    # ОБУЧЕНИЕ НА СГЕНЕРИРОВАННЫХ ИЗОБРАЖЕНИЯХ
    noise = np.random.normal(0, 1, (batch_size, z_dim))
    gen_imgs = generator.predict(noise)
    self.discriminator.train_on_batch(gen_imgs, fake) ❷

def train_generator(batch_size):

    valid = np.ones((batch_size,1))
    noise = np.random.normal(0, 1, (batch_size, z_dim))
    self.model.train_on_batch(noise, valid) ❸

epochs = 2000
batch_size = 64

for epoch in range(epochs):

    train_discriminator(x_train, batch_size)
    train_generator(batch_size)
```

❶ Один цикл обучения дискриминатора включает первый этап обучения на пакете истинных изображений с ответом 1...

❷ ...и второй этап обучения на пакете сгенерированных изображений с ответом 0.

❸ Один цикл обучения генератора включает обучение на пакете сгенерированных изображений с ответами 1. Поскольку веса дискриминатора зафиксированы, изменяться будут только веса генератора, в направлении, помогающем генерировать изображения, которые с большей вероятностью будут восприняты дискриминатором как истинные (то есть для которых дискриминатор будет возвращать прогноз, близкий к 1).

Спустя несколько эпох дискриминатор и генератор найдут равновесие, позволяющее генератору начать улавливать значимую информацию от дискриминатора, после чего качество генерируемых изображений станет улучшаться (рис. 4.8).

Наблюдая за изображениями, созданными генератором в разные эпохи обучения (рис. 4.9), легко заметить, что генератор постепенно набирается опыта в создании изображений, похожих на изображения из обучающего набора.

Немного удивительно, что нейронная сеть способна преобразовать случайный шум во что-то значимое. Мы не передали модели никаких дополнительных признаков, кроме простых пикселов, поэтому она должна самостоятельно выявить высокоуровневые признаки, такие как *горб*, *нога* или *голова*. Наивные байесовские модели (см. главу 1) не способны достичь такого уровня сложности, потому что не могут моделировать взаимозависимости между пикселами, которые имеют решающее значение для формирования высокоуровневых признаков. Еще одно требование к успешной генеративной модели — она не должна воспроизводить изображения из обучающего набора. Чтобы убедиться, что наша модель соблюдает это требование, можно найти изображение из обучающего набора, наиболее близкое к конкретному сгенерированному примеру, и определить расстояние между ними. В качестве меры расстояния можно использовать расстояние L1, определяемое так:

```
def l1_compare_images(img1, img2):
    return np.mean(np.abs(img1 – img2))
```

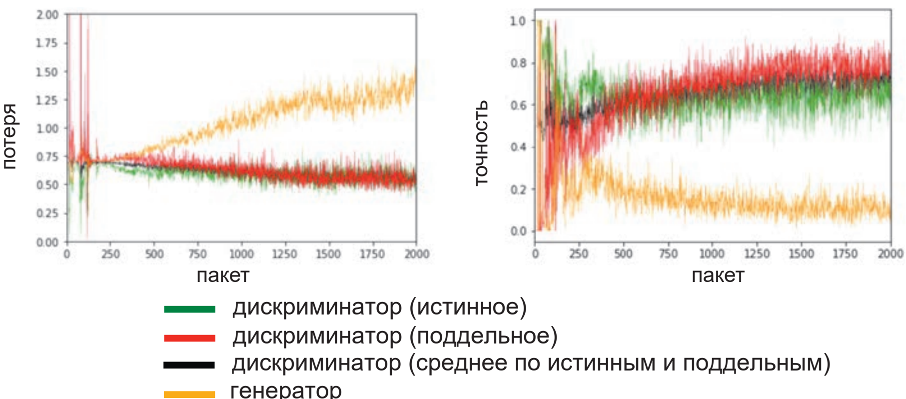



**Рис. 4.8.** Изменение потери точности дискриминатора и генератора в ходе обучения

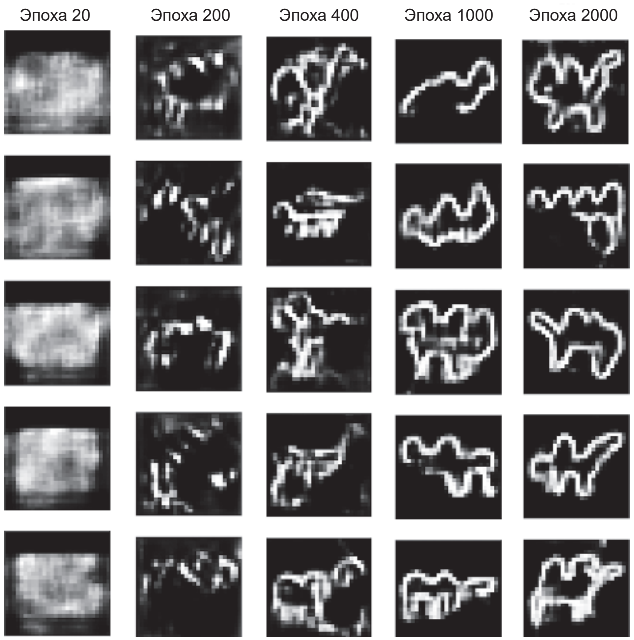



**Рис. 4.9.** Изображения, созданные генератором в разные эпохи обучения

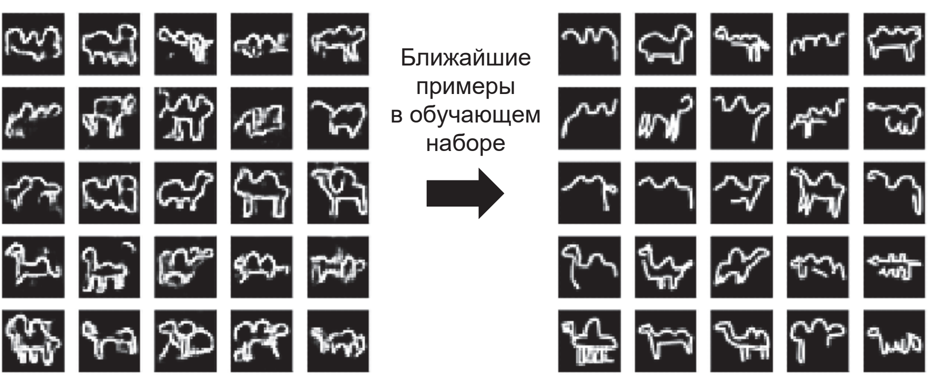



**Рис. 4.10.** Самые близкие друг к другу сгенерированные и обучающие изображения

На рис. 4.10 показаны самые близкие друг к другу сгенерированные и обучающие изображения. Как видите, несмотря на определенное сходство, сгенерированные и обучающие изображения не идентичны. Кроме того, наша сеть может даже завершить некоторые незаконченные рисунки, например, добавив ноги или голову. Это доказывает, что генератор выявил соответствующие высокоуровневые признаки и смог сгенерировать примеры, отличные от тех, которые он уже видел.

## Проблемы генеративно-состязательных сетей

Появление генеративно-состязательных сетей стало крупным шагом в развитии генеративного моделирования, но они очень сложны в обучении. Рассмотрим наиболее типичные проблемы, возникающие при обучении GAN, а затем исследуем некоторые изменения в архитектуре GAN, помогающие устранить многие из этих проблем.

### Колебания потерь

Потери в дискриминаторе и генераторе могут неожиданно начать колебаться в широком диапазоне, вместо того чтобы стабилизироваться. Как правило, между пакетами наблюдаются небольшие колебания потерь, но в процессе обучения потери должны стабилизироваться или постепенно увеличиваться

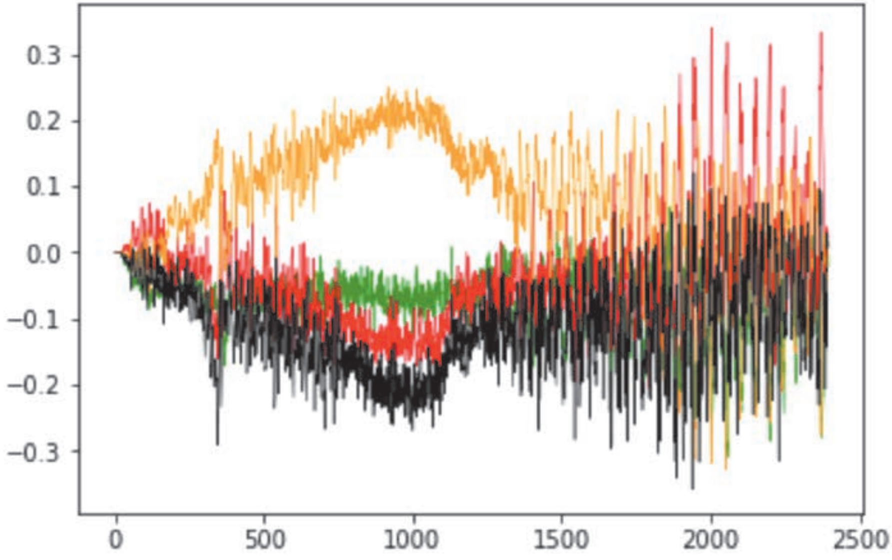



**Рис. 4.11.** Колебания потерь в нестабильной сети GAN

или уменьшаться (см. рис. 4.8), что обеспечивает сходимость сети GAN и улучшение ее генеративных способностей. На рис. 4.11 показан пример GAN, в которой потери дискриминатора и генератора начали выходить из-под контроля где-то в районе пакета 1400. Трудно предсказать, произойдет ли это когда-либо, потому что обычным генеративно-состязательным сетям свойственны такого рода нестабильности.

## Коллапс модели

Коллапс модели возникает, когда генератор находит небольшое количество образцов, обманывающих дискриминатор, из-за чего оказывается не способен сгенерировать никаких других примеров, кроме этого ограниченного набора. Подумаем, как это может произойти. Предположим, что мы обучаем генератор на нескольких пакетах, не обновляя дискриминатор между ними. В этом случае генератор будет стремиться найти единственное наблюдение (также известное как *форма* (mode)), которое всегда обманывает дискриминатор, и начинает сопоставлять каждую точку в скрытом пространстве с этим наблюдением. В результате градиент функции потерь падает почти до нуля. Если затем попытаться возобновить обучение дискриминатора, чтобы

прекратить его одурачивание этой единственной точкой, то генератор просто найдет другую форму, обманывающую дискриминатор, поскольку он уже стал нечувствителен к его вкладу и, следовательно, не имеет стимулов для увеличения разнообразия своих результатов (рис. 4.12).

## Неинформативные потери

Поскольку целью модели глубокого обучения является минимизация функции потерь, естественно полагать, что чем меньшее значение дает функция потерь генератора, тем выше качество получаемых им изображений. Но потери генератора оцениваются по текущему дискриминатору, который постоянно совершенствуется, поэтому нельзя сравнивать потери, полученные в разные моменты в процессе обучения. Действительно, потери генератора на рис. 4.8 со временем увеличиваются, хотя качество изображений явно улучшается. Отсутствие корреляции между потерями генератора и качеством изображений иногда затрудняет контроль над процессом обучения GAN.

## Гиперпараметры

Даже в простых генеративно-состязательных сетях имеется большое количество гиперпараметров, доступных для настройки. Наряду с общей архитектурой дискриминатора и генератора необходимо учитывать параметры, управляющие пакетной нормализацией, прореживанием, скоростью обучения, слоями активации, сверточными фильтрами, размером ядра, величиной шага, размером пакета и размером скрытого пространства. Генеративно-состязательные сети очень чувствительны даже к самым незначительным изменениям этих параметров, и нередко поиск подходящего набора параметров выполняется методом проб и ошибок, а не в соответствии с установленным набором рекомендаций. Вот почему так важно понимать особенности работы генеративно-состязательных сетей и знать, как интерпретировать функцию потерь — чтобы осмысленно корректировать гиперпараметры, которые могут улучшить стабильность модели.

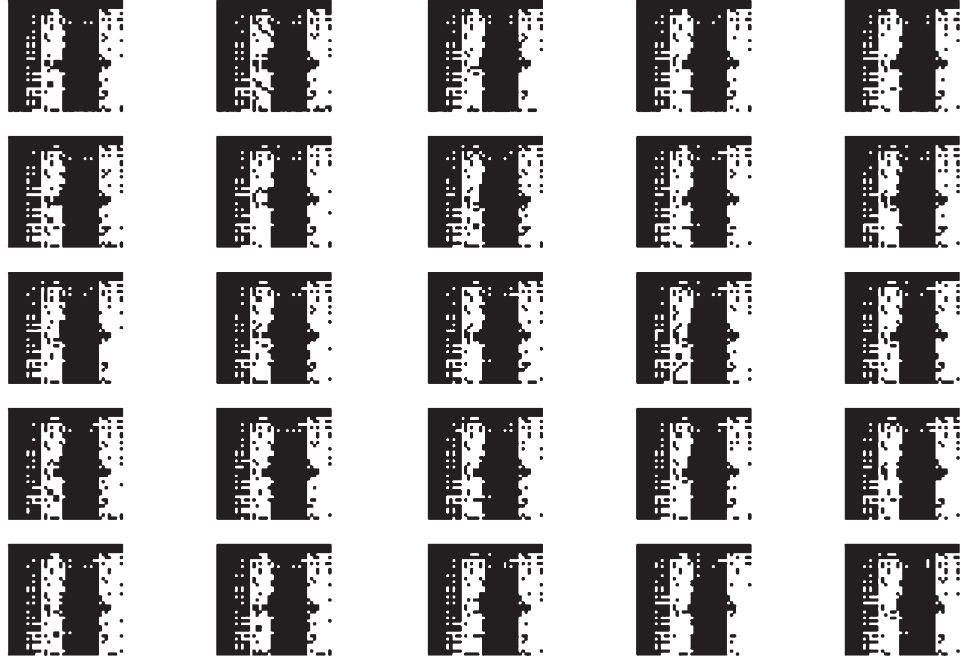


**Рис. 4.12.** Коллапс модели

## Решение проблем генеративно-состязательных сетей

За последние годы было предложено несколько ключевых улучшений, позволивших значительно повысить общую стабильность моделей GAN и уменьшить вероятность некоторых проблем, перечисленных выше, таких как коллапс модели. В оставшейся части этой главы мы рассмотрим два таких улучшения: Wasserstein GAN (WGAN) и Wasserstein GAN – Gradient Penalty (WGAN-GP). Оба являются незначительными изменениями в инфраструктуре, исследованной ранее, причем последнее улучшение в настоящее время считается лучшим из доступных на сегодняшний день для обучения наиболее совершенных GAN.

## Генеративно-состязательные сети с функцией потерь Вассерштейна

Появление генеративно-состязательных сетей с функцией потерь Вассерштейна (Wassertein GAN, WGAN) стало одним из первых больших шагов к стабилизации обучения генеративно-состязательных сетей.[1] Добавив несколько изменений, авторам этого подхода удалось показать, как получить генеративно-состязательную сеть, которая обладает следующими двумя свойствами (цитата из статьи):

- осмысленные значения потерь, коррелирующие со сходимостью генератора и качеством образцов;
- повышенная стабильность процесса оптимизации.

В частности, в статье представлена новая функция потерь для дискриминатора и генератора. Использование этой функции потерь вместо бинарной перекрестной энтропии дает более стабильную сходимость GAN. Математическое объяснение этой функции выходит за рамки книги, но в интернете есть несколько превосходных ресурсов, где дается обоснование перехода к ней.

Рассмотрим определение функции потерь Вассерштейна (Wasserstein).

### Функция потерь Вассерштейна

Для начала вспомним, как выглядит бинарная перекрестная энтропия — функция, которую мы использовали для обучения дискриминатора и генератора генеративно-состязательной сети:

*Бинарная перекрестная энтропия*

$$-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log\left(p_i\right)+\left(1-y_i\right)\log\left(1-p_i\right)\right).$$

В процессе обучения дискриминатора $D$ мы вычисляем потери, сравнивая оценки для реальных изображений $p_i = D(x_i)$ с ответом $y_i = 1$ и оценки для сгенерированных изображений $p_i = D(G(z_i))$ с ответом $y_i = 0$. Минимизацию функции потерь для дискриминатора GAN можно записать так:

[1] Мартин Аржовски (Martin Arjovsky) и др., «Wasserstein GAN», 26 января 2017, https://arxiv.org/pdf/1701.07875.pdf.

*Минимизация функции потерь для дискриминатора GAN*

$$\min_D -\left(\mathrm{E}_{x\sim p_X}\left[\log D(x)\right]+\mathrm{E}_{z\sim p_Z}\left[\log\left(1-D\left(G(z)\right)\right)\right]\right).$$

В процессе обучения генератора $G$ мы вычисляем потери, сравнивая оценки для сгенерированных изображений $p_i = D(G(z_i))$ с ответом $y_i = 1$. Соответственно, минимизацию функции потерь для генератора GAN можно записать так:

$$\min_G -\left(\mathrm{E}_{z\sim p_Z}\left[\log D\left(G(z)\right)\right]\right).$$

Сравнение с функцией потерь Вассерштейна позволяет заключить следующее. Во-первых, функция потерь Вассерштейна требует использовать метки $y_i = 1$ и $y_i = -1$ вместо 1 и 0. Из последнего слоя дискриминатора нужно убрать функцию активации *sigmoid* с тем, чтобы оценки $p_i$ больше не ограничивались диапазоном [0, 1], и теперь оценки могут принимать любые значения в диапазоне $[-\infty, \infty]$. По этой причине дискриминатор в WGAN обычно называют *критиком*. Функция потерь Вассерштейна тогда определяется следующим образом:

$$-\frac{1}{n}\sum_{i=1}^{n}(y_i p_i).$$

В процессе обучения критика $D$ мы вычисляем потери, сравнивая оценки для реальных изображений $p_i = D(x_i)$ с ответом $y_i = 1$ и оценки для сгенерированных изображений $p_i = D(G(z_i))$ с ответом $y_i = -1$. То есть минимизацию функции потерь для критика WGAN можно записать так:

$$\min_D -\left(\mathrm{E}_{x\sim p_X}\left[D(x)\right]-\mathrm{E}_{z\sim p_Z}\left[D\left(G(z)\right)\right]\right).$$

Иными словами, критик WGAN пытается максимизировать разницу между оценками для реальных и сгенерированных изображений, при этом реальные изображения получают более высокую оценку. В процессе обучения генератора WGAN мы вычисляем потери, сравнивая оценки для сгенерированных изображений $p_i = D(G(z_i))$ с ответом $y_i = 1$. Значит, минимизацию функции потерь для генератора WGAN можно записать так:

$$\min_G -\left(\mathrm{E}_{z\sim p_Z}\left[D\left(G(z)\right)\right]\right).$$

Компилируя модели, обучающие критика и генератор WGAN, отметим, что вместо бинарной перекрестной энтропии хотим использовать функцию потерь Вассерштейна (листинг 4.6). Добавим, что для WGAN обычно используются меньшие скорости обучения.

**Листинг 4.6.** Компиляция моделей, обучающих критика и генератор

```
def wasserstein(y_true, y_pred):
    return -K.mean(y_true * y_pred)

critic.compile(
    optimizer= RMSprop(lr=0.00005)
    , loss = wasserstein
)

model.compile(
    optimizer= RMSprop(lr=0.00005)
    , loss = wasserstein
)
```

## Ограничение Липшица

Вместо применения функции активации *sigmoid*, ограничивающей выходное значение обычным диапазоном [0, 1], мы позволили критику возвращать любые оценки в диапазоне [–∞, ∞]. По этой причине величина потери Вассерштейна может получиться очень большой, что вызывает беспокойство — обычно следует избегать появления слишком больших чисел в нейронных сетях!

Фактически авторы статьи о WGAN утверждают, что при использовании функции потерь Вассерштейна также необходимо наложить дополнительное ограничение на критика. В частности, критик должен быть *одномерной липшицевой непрерывной функцией*. Остановимся на этом подробнее. Критик — это функция $D$, преобразующая изображение в оценку реальности. Мы говорим, что это одномерная липшицева функция, если она удовлетворяет следующему неравенству для любых двух входных изображений, $x_1$ и $x_2$:

$$\frac{\left|D(x_1) - D(x_2)\right|}{\left|x_1 - x_2\right|} \leq 1.$$

Здесь $x_1 - x_2$ — это средняя попиксельная абсолютная разница между двумя изображениями, а $|D(x_1) - D(x_2)|$ — абсолютная разница между оценками критика. По сути, мы ограничиваем скорость, с которой могут меняться

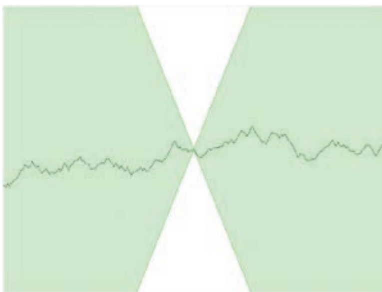


**Рис. 4.13.** Непрерывная функция Липшица — существует такой двойной конус, что где бы он ни находился на прямой, функция всегда остается полностью вне конуса[1]

оценки критика между двумя изображениями (то есть абсолютное значение градиента нигде не должно превышать единицу). Это верно для одномерной непрерывной функции Липшица (рис. 4.13) — ни в одной точке линия не попадает в область конуса, где бы вы ни поместили конус на линии. Иначе говоря, скорость, с которой линия может подниматься или опускаться в любой точке, ограничена.

Желающим глубже вникнуть в математическое обоснование, почему функция потерь Вассерштейна может дать положительные результаты, только когда применяется это ограничение, Джонатан Хуэй (Jonathan Hui) предлагает отличное объяснение в своей статье (http://bit.ly/2MwS8rc).

## Усечение весов

Авторы статьи о WGAN показывают, как реализовать ограничение Липшица усечением весов критика до диапазона [–0,01, 0,01] после каждого обучающего пакета. Добавить усечение весов в нашу функцию обучения критика WGAN можно так, как показано в листинге 4.7.

**Листинг 4.7.** Обучение критика WGAN

```
def train_critic(x_train, batch_size, clip_threshold):

    valid = np.ones((batch_size,1))
    fake = -np.ones((batch_size,1))

    # ОБУЧЕНИЕ НА РЕАЛЬНЫХ ИЗОБРАЖЕНИЯХ
    idx = np.random.randint(0, x_train.shape[0], batch_size)
```

[1] Источник: Википедия, http://bit.ly/2Xufwd8

```
    true_imgs = x_train[idx]
    self.critic.train_on_batch(true_imgs, valid)

    # ОБУЧЕНИЕ НА СГЕНЕРИРОВАННЫХ ИЗОБРАЖЕНИЯХ
    noise = np.random.normal(0, 1, (batch_size, self.z_dim))
    gen_imgs = self.generator.predict(noise)
    self.critic.train_on_batch(gen_imgs, fake)

    for l in critic.layers:
        weights = l.get_weights()
        weights = [np.clip(w, -clip_threshold, clip_threshold) for w in weights]
        l.set_weights(weights)
```

## Обучение WGAN

При использовании функции потерь Вассерштейна критик обучается сходимости, что обеспечивает правильное изменение градиентов, используемых для обновления генератора. Этим WGAN отличается от стандартной сети GAN, где важно не допустить, чтобы дискриминатор стал слишком сильным, и избежать затухания градиентов. Таким образом, применение потери Вассерштейна устраняет одну из ключевых сложностей, возникающих при обучении GAN, — поиск правильного баланса между обучением дискриминатора и генератора. В случае с WGAN можно несколько раз обучить критика между обновлениями генератора, чтобы убедиться, что он близок к сходимости. Типичное соотношение — пять обновлений критика на одно обновление генератора (листинг 4.8).

**Листинг 4.8.** Обучение WGAN

```
for epoch in range(epochs):

    for _ in range(5):
        train_critic(x_train, batch_size = 128, clip_threshold = 0.01)

    train_generator(batch_size)
```

Теперь мы охватили все основные различия между GAN и WGAN. Напомню, что:

- WGAN использует функцию потерь Вассерштейна;
- при обучении WGAN в качестве меток используются: 1 для реальных изображений и –1 для поддельных;

- в последнем слое критика в WGAN не требуется использовать функцию активации *sigmoid*;
- веса критика усекаются после каждого их обновления;
- перед каждым обновлением генератора выполняется несколько циклов обучения критика.

## Анализ WGAN

Вы можете обучить свою сеть WGAN, использовав код из блокнота 04_02_wgan_cifar_train.ipynb (репозиторий с примерами для книги). Этот код обучает сеть WGAN генерировать изображения лошадей на примерах из набора CIFAR-10, который мы использовали в главе 2. На рис. 4.14 показаны некоторые образцы, сгенерированные сетью WGAN.

Очевидно, эта задача сложнее предыдущей, где мы генерировали изображения ганимала, но сеть WGAN прекрасно справилась с ней, сумев определить ключевые характеристики изображений лошадей (ноги, небо, трава, гнедой окрас, тень и т. д.). Кроме цвета, WGAN выявила в обучающем наборе множество различных углов, форм и фонов. Да, качество изображений далеко не идеальное, но сам факт, что наша сеть WGAN смогла точно выявить высокоуровневые признаки, характерные для цветных фотографий лошадей, вселяет определенный оптимизм.

Один из главных недостатков сети WGAN – снижение ее способности к обучению из-за усечения весов критика. В статье о WGAN об этом сказано следующее: «Усечение весов – далеко не лучший способ применения ограничения Липшица». Сильный критик имеет решающее значение для успеха WGAN, потому что без точных градиентов генератор не сможет научиться адаптировать свои веса для получения лучших образцов. Поэтому другие исследователи продолжили поиск альтернативных способов применения ограничения Липшица и увеличения способности WGAN выявлять сложные признаки. Один из таких прорывных способов мы и рассмотрим в следующем разделе.

**Рис. 4.14.** Образцы, созданные генератором WGAN, обученным на изображениях лошадей

## WGAN-GP

Одним из последних развитий идеи WGAN является структура Wasserstein GAN–Gradient Penalty (WGAN-GP).[1] Генератор WGAN-GP определяется и компилируется точно так же, как и генератор WGAN. Изменения касаются только определения и компиляции критика. В целом требуется внести три изменения в определение критика WGAN, чтобы преобразовать его в критика WGAN-GP:

- добавить в функцию потерь член штрафования градиента;
- убрать усечение весов;
- не использовать слои пакетной нормализации.

Для начала посмотрим, как добавить штрафование градиента в функцию потерь. В статье, описывающей сети этого вида, авторы предлагают альтернативный способ применения ограничения Липшица к критику. Вместо усечения весов они применяют ограничение напрямую, включив в функцию потерь член, определяющий величину штрафа в зависимости от отклонения нормы градиента критика от 1. Согласимся, это — более естественный

[1] Ишаан Гулраджани (Ishaan Gulrajani) и др., «Improved Training of Wasserstein GANs», 31 марта 2017, https://arxiv.org/abs/1704.00028.

способ применения ограничения, обеспечивающий бо́льшую стабильность процессу обучения.

## Функция потерь штрафа за градиент

На рис. 4.15 показана диаграмма процесса обучения критика. Если сравнить ее с диаграммой процесса обучения дискриминатора на рис. 4.7, то можно увидеть, что ключевым дополнением в общую функцию потерь, кроме функции потерь Вассерштейна, является штраф за градиент между реальными и поддельными изображениями.

Функция потерь штрафа за градиент измеряет квадрат разности между нормой градиента прогнозов для входных изображений и 1. Естественно, модель будет стремиться найти веса, гарантирующие минимальный штраф за градиент, стараясь тем самым соответствовать ограничению Липшица.

Однако было бы очень сложно вычислять этот градиент повсюду в процессе обучения, поэтому в WGAN-GP он оценивается лишь по нескольким точкам. Для сохранения баланса мы используем набор интерполированных изображений, лежащих в случайно выбранных точках вдоль линий, соединяющих пары реальных и поддельных изображений в пакете (рис. 4.16).

Для выполнения этой интерполяции в Keras можно создать слой `RandomWeightedAverage`, наследующий встроенный слой `_Merge`:

```
class RandomWeightedAverage(_Merge):
    def __init__(self, batch_size):
        super().__init__()
        self.batch_size = batch_size

    def _merge_function(self, inputs):
        alpha = K.random_uniform((self.batch_size, 1, 1, 1)) ❶
        return (alpha * inputs[0]) + ((1 – alpha) * inputs[1]) ❷
```

❶ Каждому изображению в пакете присваивается случайное число в диапазоне от 0 до 1, сохраняемое как вектор `alpha`.

❷ Слой возвращает набор интерполированных изображений, лежащих на линиях, которые соединяют реальные изображения (`inputs[0]`) с поддельными (`inputs[1]`), и взвешенных по значению `alpha` для каждой пары.

Функция `gradient_penalty_loss`, показанная в листинге 4.9, возвращает квадрат разности между градиентом, вычисленным в точке интерполяции, и 1.

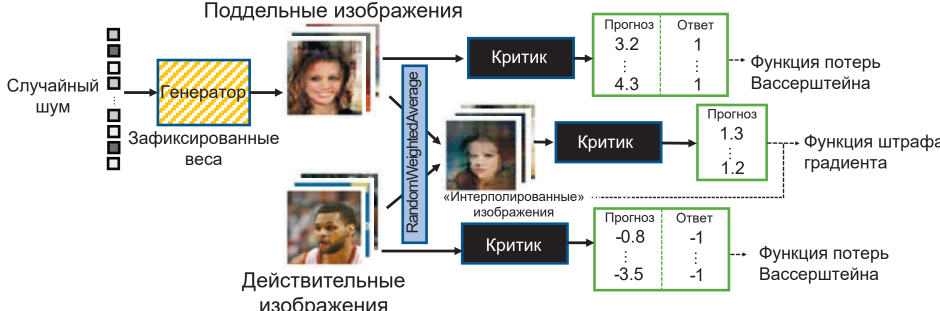



**Рис. 4.15.** Процесс обучения критика WGAN-GP

**Листинг 4.9.** Функция потерь штрафа за градиент

```
def gradient_penalty_loss(y_true, y_pred, interpolated_samples):

    gradients = K.gradients(y_pred, interpolated_samples)[0] ❶

    gradient_l2_norm = K.sqrt(
        K.sum(
            K.square(gradients),
            axis=[1:len(gradients.shape)]
            )
        )
    ) ❷

    gradient_penalty = K.square(1 – gradient_l2_norm)
    return K.mean(gradient_penalty) ❸
```

❶ Функция `gradients` в Keras вычисляет градиенты прогнозов для интерполированных изображений (`y_pred`) в отношении входа (`interpolated_samples`).

❷ Вычисляется норма L2 для этого вектора (то есть его евклидова длина).

❸ Функция возвращает квадрат расстояния между нормой L2 и 1.

Теперь слой `RandomWeightedAverage`, который находит интерполяцию между двумя изображениями, и функцию `gradient_penalty_loss`, вычисляющую потери градиентов для интерполированных изображений, можно использовать в процессе компиляции модели критика.

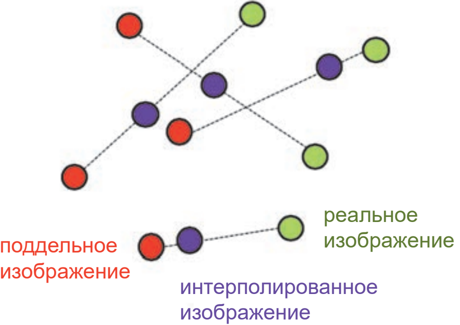



**Рис. 4.16.** Интерполяция изображений

В примере WGAN мы напрямую скомпилировали критика так, чтобы он оценивал реальность данного изображения. Чтобы скомпилировать критика WGAN-GP, нужно использовать интерполированные изображения в функцию потерь, но Keras поддерживает пользовательские функции потерь только с двумя параметрами: прогнозами и истинными метками. Чтобы обойти эту проблему, используем функцию `partial` из стандартной библиотеки Python (листинг 4.10).

**Листинг 4.10.** Компиляция критика WGAN-GP

```
from functools import partial

### КОМПИЛЯЦИЯ МОДЕЛИ КРИТИКА

self.generator.trainable = False ❶

real_img = Input(shape=self.input_dim) ❷
z_disc = Input(shape=(self.z_dim,))
fake_img = self.generator(z_disc)

fake = self.critic(fake_img) ❸
valid = self.critic(real_img)

interpolated_img = RandomWeightedAverage(self.batch_size)([real_img,
                                                           fake_img]) ❹
validity_interpolated = self.critic(interpolated_img)

partial_gp_loss = partial(self.gradient_penalty_loss,
                          interpolated_samples = interpolated_img) ❺
partial_gp_loss.__name__ = 'gradient_penalty' ❻
```

```
self.critic_model = Model(inputs=[real_img, z_disc],
                          outputs=[valid, fake, validity_interpolated]) ❼

self.critic_model.compile(
    loss=[self.wasserstein,self.wasserstein, partial_gp_loss]
    ,optimizer=Adam(lr=self.critic_learning_rate, beta_1=0.5)
    ,loss_weights=[1, 1, self.grad_weight]
) ❽
```

❶ Зафиксировать веса генератора. Это необходимо, потому что генератор является частью модели, которая используется для обучения критика, и интерполированные изображения теперь активно используются в функции потерь.

❷ Модель имеет два входа: пакет реальных изображений и набор случайно сгенерированных чисел, которые используются для создания пакета поддельных изображений.

❸ Реальные и поддельные изображения передаются критику для вычисления потери Вассерштейна.

❹ Слой `RandomWeightedAverage` создает интерполированные изображения, которые затем передаются критику.

❺ Библиотека Keras поддерживает функции потерь только с двумя параметрами — предсказаниями и истинными метками, — и чтобы организовать передачу интерполированных изображений в нашу функцию `gradient_penalty_loss`, мы определяем свою функцию потерь `partial_gp_loss`, использовав функцию `partial` из стандартной библиотеки Python.

❻ Keras требует, чтобы функции были именованными.

❼ Модель, которая обучает критика, имеет два входа (пакет реальных и пакет случайно сгенерированных поддельных изображений) и три выхода: 1 — для реальных изображений, –1 — для поддельных изображений и фиктивный вектор 0, который на самом деле не используется, но его требует библиотека Keras, потому что каждая функция потерь должна отображаться в выход. Поэтому был создан фиктивный вектор 0 для функции `partial_gp_loss`.

❽ Критик компилируется с тремя функциями потерь: две функции потерь Вассерштейна для реальных и поддельных изображений и функция потерь штрафа за градиент. Общая потеря вычисляется как сумма этих трех потерь,

причем потеря за градиент взвешивается множителем 10, в соответствии с рекомендациями в оригинальной статье. Мы используем оптимизатор Adam, который считается лучшим оптимизатором для моделей WGAN-GP.

**ПАКЕТНАЯ НОРМАЛИЗАЦИЯ В WGAN-GP**

Прежде чем перейти к созданию WGAN-GP, важно отметить, что в критике не должна использоваться пакетная нормализация. Причина в том, что пакетная нормализация создает корреляцию между изображениями в одном пакете, что снижает эффективность штрафа за градиент. Как показали эксперименты, сети WGAN-GP способны давать превосходные результаты даже без пакетной нормализации в критике.

## Анализ WGAN-GP

Для обучения модели WGAN-GP на наборе данных CelebA с фотографиями лиц знаменитостей запустим блокнот 04_03_wgangp_faces_train.ipynb из репозитория с примерами к книге. Сначала рассмотрим несколько случайно выбранных образцов, созданных генератором после обучения на 3000 пакетах (рис. 4.17).

**Рис. 4.17.** Примеры изображений, сгенерированных сетью WGAN-GP после обучения на наборе данных CelebA

Как видим, модель сумела определить существенные высокоуровневые признаки лиц и не проявила признаков коллапса. Обратим внимание и на то, как эволюционировали функции потерь модели с течением времени (рис. 4.18) — функции потерь обоих компонентов, дискриминатора и генератора, остаются стабильными и постепенно сходятся.

Если сравнить вывод сети WGAN-GP с выводом вариационного автокодировщика, то можно увидеть, что генеративно-состязательная сеть создает, как правило, более четкие изображения, особенно это касается границы между волосами и фоном. Вариационные автокодировщики склонны создавать более мягкие изображения, с размытыми границами цвета, тогда как генеративно-состязательные сети создают более четкие и контрастные изображения. Столь же верно и то, что генеративно-состязательные сети, как правило, труднее поддаются обучению, чем вариационные автокодировщики, и для достижения удовлетворительного качества требуется больше времени. Тем не менее большинство современных генеративных моделей основано на генеративно-состязательных сетях, потому что обучение крупномасштабных GAN на графических процессорах в течение длительного времени дает более впечатляющие результаты.

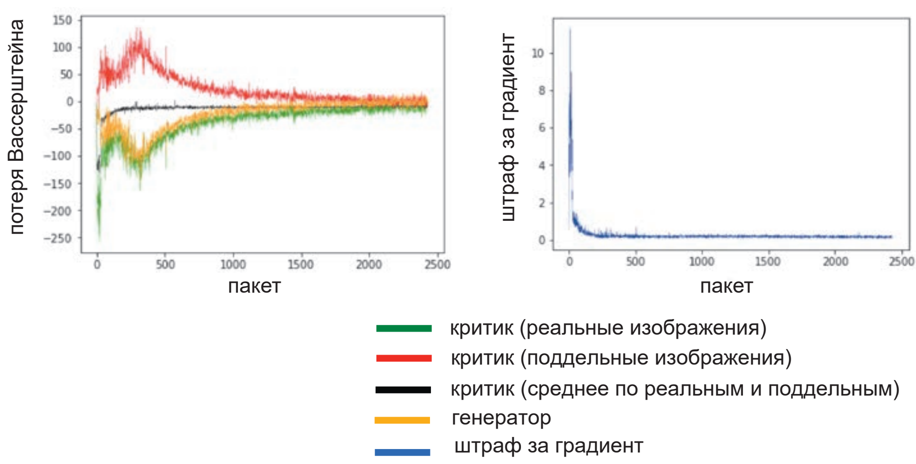



**Рис. 4.18.** Функции потерь в WGAN-GP

## Итоги

В этой главе мы рассмотрели три разновидности генеративно-состязательных сетей: от самых простых GAN до Wasserstein GAN (WGAN) и самых современных WGAN-GP.

Все генеративно-состязательные сети имеют архитектуру «генератор против дискриминатора (или критика)», где дискриминатор пытается «заметить разницу» между реальными и поддельными изображениями, а генератор стремится обмануть дискриминатор. При сбалансированном обучении этих двух соперников генератор GAN может постепенно научиться производить наблюдения, похожие на наблюдения в обучающем наборе.

Простым генеративно-состязательным сетям свойственно несколько проблем, включая коллапс модели и нестабильность обучения. Функция потерь Вассерштейна избавляет от многих из этих проблем, делая обучение WGAN более предсказуемым и надежным. Естественное развитие WGAN — модель WGAN-GP, включающая в процесс обучения одномерную непрерывную функцию Липшица в виде члена в функции потерь, чтобы подтянуть норму градиента к 1.

Наконец, применив новый подход к задаче генерации лиц, мы увидели, как, выбирая точки из стандартного нормального распределения, можно получать новые лица. Процесс выборки очень похож на использовавшийся в вариационном автокодировщике, но лица, созданные сетью GAN, существенно отличаются — они часто более контрастные и имеют более четкие границы между различными частями. При обучении на большом количестве графических процессоров это свойство позволяет генеративно-состязательным сетям добиваться впечатляющих результатов и поднимает генеративное моделирование на новый, более высокий уровень.

В целом, как видим, генеративно-состязательные сети обладают огромной гибкостью и с успехом могут применяться во многих практических направлениях. Один из вариантов применения мы и рассмотрим в следующей главе с тем, чтобы выяснить, как можно научить машины рисовать.

# Часть II

# Учим машины рисовать, писать, сочинять музыку и играть в игры

В **части I** была представлена область глубокого генеративного обучения и проанализированы два наиболее важных достижения последних лет: вариационные автокодировщики и генеративно-состязательные сети. Остальная часть этой книги представляет набор тематических исследований, демонстрирующих применение методов генеративного моделирования для решения конкретных задач. Следующие три главы посвящены трем основным столпам человеческого творчества: живописи, литературному творчеству и сочинению музыки.

В **главе 5** мы рассмотрим два приема, относящихся к машинному художественному творчеству. Сначала речь пойдет о сети CycleGAN, которая, как следует из названия, основана на архитектуре GAN и позволяет научить модель преобразовывать фотографии в картины в определенном стиле (и наоборот). Затем обратим внимание на прием нейронной передачи стиля, реализованный во многих приложениях для редактирования фотографий, позволяющий перенести стиль картины на фотографию, чтобы создать впечатление, что картина принадлежит кисти того же художника.

В **главе 6** мы возьмемся за задачу обучения машины литературному творчеству — задачу, которой присущи многие сложности, связанные и с созданием изображений. В этой главе описывается архитектура рекуррентной нейронной сети (Recurrent Neural Network, RNN), позволяющая преодолевать проблемы, связанные с созданием последовательностей данных. Мы также увидим, как работает архитектура кодер-декодер, и построим простой генератор вопросов и ответов.

В **главе 7** рассматривается задача машинного обучения генерации музыки, которая, хотя и является задачей генерации последовательностей, сталкивается с дополнительными сложностями, такими как моделирование музыкальной тональности и ритма. Здесь мы увидим, что многие методы, применявшиеся для генерации текста,

с успехом могут применяться и в этой области, а также исследуем архитектуру глубокого обучения, известную как MuseGAN, которая применяет идеи из главы 4 (о генеративно-состязательных сетях) для сочинения музыки.

**Глава 8** демонстрирует, как генеративные модели могут использоваться в других областях машинного обучения, например в обучении с подкреплением. В этой главе представлена одна из самых захватывающих работ, опубликованных за последние годы, в которой авторы показывают, как использовать генеративную модель в качестве среды обучения агента и позволить ему *мечтать* о возможных будущих сценариях и, в рамках своей концептуальной модели окружающей среды, представлять, что произойдет, если он предпримет определенные действия.

Наконец, **глава 9** обобщает текущую картину генеративного моделирования и методы, представленные в этой книге. В этой главе мы заглянем в будущее и попытаемся выяснить, как самые современные методы могут изменить наш взгляд на творчество и сможем ли мы когда-нибудь создать искусственную сущность, способную создавать творческие произведения, неотличимые от работ, созданных художниками, писателями и композиторами.

ГЛАВА 5

# Рисование

До сих пор мы исследовали способы обучения моделей созданию новых образцов на основе только обучающего набора данных, которые модель должна имитировать. Эти способы применены к нескольким наборам данных. В результате вариационные автокодировщики и генеративно-состязательные сети способны установить соответствие между скрытым пространством и пространством исходных пикселов. Выбрав точку из распределения в скрытом пространстве, можно с помощью генеративной модели получить для нее новое изображение в пространстве пикселов. Отметим, что все рассмотренные ранее примеры создают новые наблюдения с нуля, не получая никаких других входных данных, кроме случайного вектора, взятого из скрытого пространства.

Другой областью применения генеративных моделей является *передача стиля*. В этом случае наша цель — построить модель, способную преобразовывать входное *базовое изображение* так, чтобы создать впечатление, что оно принадлежит той же коллекции, что и данный набор *изображений с образцами стиля*. Этот метод имеет явные коммерческие применения и в настоящее время используется в системах компьютерной графики, компьютерных играх и приложениях для мобильных телефонов (см. примеры на рис. 5.1).

Целью передачи стиля является не моделирование базового распределения изображений с образцами стиля, а извлечение из этих изображений стилистических компонентов и их внедрение в базовое изображение. Совершенно понятно, что нельзя объединить базовое изображение с изображением, представляющим образец стиля, посредством интерполяции, потому что содержимое изображения стиля будет просвечивать сквозь базовое изображение, а цвета станут мутными и размытыми. Более того, образцов стиля,

**Рис. 5.1.** Примеры передачи стиля[1]

отражающих стиль художника, может быть множество, поэтому нужно найти способ, помогающий модели изучить стиль во всей коллекции изображений. Мы должны создать впечатление, что художник использовал базовое изображение как основу для создания оригинального произведения искусства, дополнив его стилевыми элементами, характерными для других произведений в его коллекции.

В этой главе вы узнаете, как построить две разные модели передачи стиля (CycleGAN и Neural Style Transfer) и с их помощью преобразовать свои фотографии и иллюстрации. А начнем мы с посещения овощного магазина, где все не так, как кажется.

## Яблоки и апельсины

Бабуля Смит и Флорида — совладелицы овощного магазина. Чтобы добиться максимальной эффективности, каждая заботится о своих полках в магазине. Например, бабуля Смит гордится своим выбором яблок, а Флорида часами раскладывает апельсины, добиваясь их идеального расположения. Обе настолько уверены в себе, что согласились на сделку: прибыль от продажи яблок пойдет на долю бабули Смит, прибыль от продажи апельсинов — на долю Флориды.

[1] Джун-Ян Чжу (Jun-Yan Zhu) и др., «Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks», 30 марта 2017, https://arxiv.org/pdf/1703.10593.pdf.

К сожалению, ни бабуля Смит, ни Флорида не намерены состязаться честно. Когда Флорида не видит, бабуля Смит пробирается к витрине с апельсинами и начинает красить их в красный цвет, чтобы они выглядели как яблоки! Флорида действует точно так же: она пытается сделать яблоки бабули Смит более похожими на апельсины, используя спрей подходящего цвета, когда та поворачивается спиной. Из-за этого покупатели часто выбирают не те фрукты, какие намеревались купить. Те, кто собирался купить апельсины, иногда по ошибке выбирают яблоки, покрашенные Флоридой, а те, кто хотел купить яблоки, по ошибке выбирают апельсины, замаскированные бабулей Смит. В конце концов прибыль за каждый фрукт суммируется и делится соответственно договоренностям — бабуля Смит теряет деньги всякий раз, когда одно из ее яблок продается как апельсин, а Флорида — когда один из ее апельсинов продается как яблоко.

После закрытия магазина обе приступают к наведению порядка на полках с фруктами. Но вместо того чтобы пытаться отмыть краску, нанесенную соперницей, обе просто красят подделки, стараясь привести их в первоначальный вид. Для них важно сделать эту работу правильно: если фрукты не будут выглядеть как нужно, то они не смогут продать их на следующий день и снова потеряют прибыль. Чтобы убедиться в правильности своих действий, они иногда проверяют свои методы на своих собственных фруктах. Флорида опрыскивает свои апельсины и проверяет, выглядят ли они точно так же, как и первоначально. Бабуля Смит проверяет свои «умения» на яблоках. Если они обнаружат очевидные несоответствия, то им придется тратить с трудом заработанную прибыль на изучение более совершенных методов (рис. 5.2).

Первое время покупатели часто ошибаются в выборе. Однако со временем они набираются опыта и учатся определять, какие фрукты были подделаны. Это вынуждает бабулю Смит и Флориду совершенствоваться в подделке фруктов друг друга, а также в восстановлении подделок до первоначального вида. Кроме того, им нужно периодически убеждаться в том, что приемы, которые они используют, не изменяют внешний вид их собственных фруктов.

После многих дней и недель этой нелепой игры они заметили, что произошло нечто удивительное. Клиенты перестали различать настоящие и поддельные фрукты. На рис. 5.3 показано, как выглядят плоды после фальсификации и восстановления, а также после тестирования.

**Рис. 5.2.** Схема подделки, восстановления и проверки фруктов в овощном магазине

**Рис. 5.3.** Образцы апельсинов и яблок в овощном магазине

## CycleGAN

Предыдущая история — это аллегория, описывающая развитие одного из ключевых направлений в генеративном моделировании, известного как передача стиля: циклически согласованную генеративно-состязательную сеть, или CycleGAN. Статья, описывающая эту сеть, является значительным шагом вперед в области передачи стиля и показывает, как обучить модель, способную копировать стиль из набора эталонных изображений в другое изображение без использования обучающего набора парных примеров.[1]

Существовавшие до этого модели передачи стиля, такие как pix2pix[2], требовали, чтобы каждое изображение в обучающем наборе существовало и в исходной, и в целевой форме. Для некоторых задач переноса стиля (например,

[1] Чжу (Zhu) и др., 2017, https://arxiv.org/pdf/1703.10593.pdf.
[2] Филипп Изола (Phillip Isola) и др., «Image-to-Image Translation with Conditional Adversarial Networks», 2016, https://arxiv.org/abs/1611.07004.

для преобразования черно-белых фотографий в цветные, карт — в спутниковые снимки) можно создать такой набор данных, но для других это просто невозможно. Например, у нас нет оригинальных фотографий пруда, с которого Моне написал свою серию «Водяные лилии», и картины кисти Пикассо с изображением Эмпайр-стейт-билдинг. По аналогии, потребовались бы огромные усилия и для того, чтобы получить фотографии лошадей и зебр, стоящих в одинаковых позах. Статья с описанием CycleGAN вышла всего через несколько месяцев после статьи, описывающей метод pix2pix, и показывает, как обучить модель решать те же задачи, но в отсутствие пар исходных и целевых изображений. На рис. 5.4 показана разница между парными и непарными наборами данных pix2pix и CycleGAN соответственно.

Если метод pix2pix позволяет моделировать изменения только в одном направлении (от исходного стиля к целевому), то CycleGAN обучает модель в обоих направлениях одновременно, благодаря чему модель учится преобразовывать исходные изображения в целевые и обратно. Это свойство обусловлено архитектурой модели, поэтому возможность преобразования в обратном направлении вы получаете автоматически.

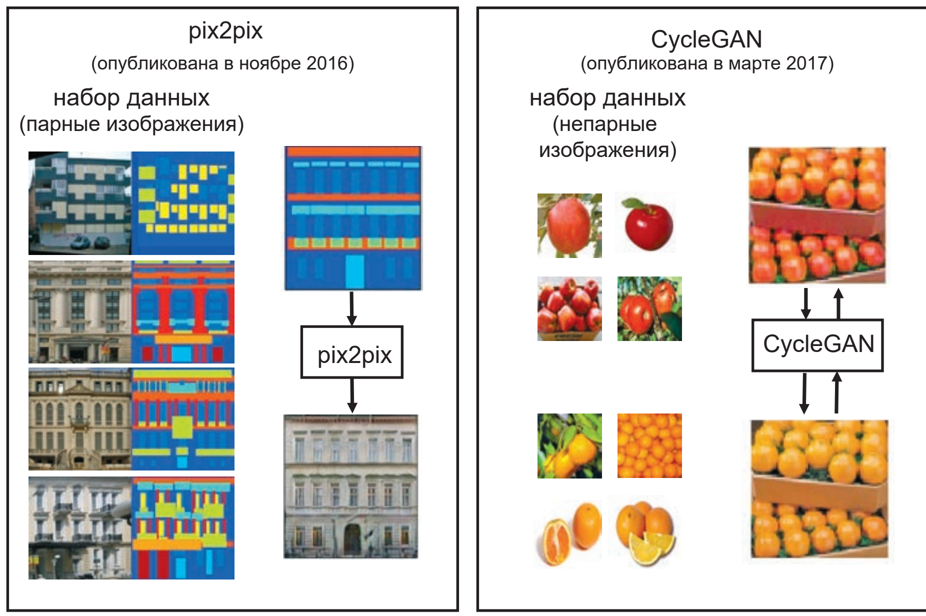



**Рис. 5.4.** Набор данных pix2pix и CycleGan и пример преобразования

Посмотрим, как сконструировать модель CycleGAN с помощью Keras. Для начала возьмем за основу предыдущий пример с яблоками и апельсинами, пройдемся по всем частям CycleGAN и поэкспериментируем с ее архитектурой. Затем применим тот же подход, чтобы создать модель, которая сможет применить стиль определенного художника к фотографии по вашему выбору.

## Ваша первая сеть CycleGAN

Большая часть следующего кода была заимствована из репозитория Keras-GAN (http://bit.ly/2Za68J2), поддерживаемого Эриком Линдером-Нореном (Erik Linder-Norén). Этот замечательный ресурс содержит множество примеров генеративно-состязательных сетей из разных книг, реализованных с помощью Keras.

Для начала вам нужно загрузить данные, которые будут использоваться для обучения. В папке, куда вы клонировали репозиторий с примерами для книги, выполните команду:

```
bash ./scripts/download_cyclegan_data.sh apple2orange
```

В загруженном наборе изображений яблок и апельсинов, которые мы будем использовать далее, данные разбиты на четыре папки: *trainA* и *testA* содержат изображения яблок, а *trainB* и *testB* — изображения апельсинов. То есть *A* представляет пространство изображений яблок, а *B* — пространство изображений апельсинов. Наша цель — обучить модель на наборах данных в папках *train* преобразовывать изображения из набора *A* в набор *B*, и наоборот. Изображения в папках *test* мы используем для проверки качества модели.

### Обзор

Сеть CycleGAN фактически состоит из четырех моделей: двух генераторов и двух дискриминаторов. Первый генератор, `G_AB`, преобразует изображения из пространства *A* в пространство *B*. Второй генератор, `G_BA`, преобразует изображения из пространства *B* в пространство *A*. В отсутствие парных изображений для обучения генераторов нужно обучить два дискриминатора, которые будут оценивать убедительность изображений, созданных генераторами. Первый дискриминатор, `d_A`, обучается различать реальные

и поддельные изображения в пространстве $A$, созданные генератором G_BA. Второй дискриминатор, d_B, обучается различать реальные и поддельные изображения в пространстве $B$, созданные генератором G_AB. Связь между четырьмя моделями показана на рис. 5.5.

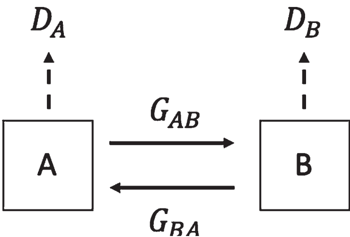



**Рис. 5.5.** Схема сети CycleGAN с четырьмя моделями[1]

Запустив блокнот 05_01_cyclegan_train.ipynb из репозитория с примерами для книги, вы можете обучить свою сеть CycleGAN. Как и в предыдущих главах, вы можете создать объект CycleGAN в блокноте (см. листинг 5.1) и поэкспериментировать с параметрами, чтобы понять, какое влияние на модель они оказывают.

**Листинг 5.1.** Определение сети CycleGAN

```
gan = CycleGAN(
        input_dim = (128,128,3)
        , learning_rate = 0.0002
        , lambda_validation = 1
        , lambda_reconstr = 10
        , lambda_id = 2
        , generator_type = 'u-net'
        , gen_n_filters = 32
        , disc_n_filters = 32
        )
```

[1] Источник: Джун-Ян Чжу (Jun-Yan Zhu) и др., 2017, https://arxiv.org/pdf/1703.10593.pdf.

Сначала посмотрим на архитектуру генераторов. Обычно в CycleGAN используются генераторы двух видов: *U-Net* и *ResNet* (Residual Network — остаточная сеть). Авторы методики pix2pix[1] использовали архитектуру U-Net, а авторы CycleGAN переключились на архитектуру ResNet. В этой главе мы попробуем обе архитектуры и начнем с U-Net.[2]

## Генераторы (U-Net)

На рис. 5.6 показана архитектура U-Net, которую мы будем использовать. Угадайте, почему она получила название U-Net.[3]

Подобно вариационному автокодировщику, сеть U-Net состоит из двух половин: понижающей разрешение — сжимающей пространственные размеры входных изображений, но расширяющей число каналов; и повышающей разрешение — увеличивающей пространственные размеры, но уменьшающей число каналов.

Однако, в отличие от вариационного автокодировщика, в U-Net между слоями с одинаковой формой в половинах, понижающих и повышающих разрешение, имеются дополнительные пропускающие связи. Вариационный автокодировщик действует линейно; данные передаются через всю сеть, от входа к выходу, от одного слоя к другому. В U-Net имеются дополнительные связи, которые пропускают часть информации по более короткому пути между слоями с одинаковой формой.

Идея заключается в следующем: с каждым последующим слоем в части сети, понижающей разрешение, модель все больше концентрируется на том, *что* изображено, и теряет информацию о том, *где* это изображено. В точке перегиба U-образной кривой карта признаков наполнится контекстной информацией о том, что находится на изображении, и почти полностью утратит информацию о том, где это находится.

Для моделей классификации этого более чем достаточно — останется только добавить завершающий полносвязанный слой и вывести вероятность

[1] Филипп Изола (Phillip Isola) и др., «Image-to-Image Translation with Conditional Adversarial Networks», 2016, https://arxiv.org/abs/1611.07004.

[2] Олаф Роннебергер (Olaf Ronneberger), Филипп Фишер (Philipp Fischer) и Томас Брокс (Thomas Brox), «U-Net: Convolutional Networks for Biomedical Image Segmentation», 18 мая 2015, https://arxiv.org/abs/1505.04597.

[3] Роннебергер (Ronneberger) и др., 2015, https://arxiv.org/abs/1505.04597.

присутствия определенного класса в изображении. Однако для решения первоначально заявленных задач (сегментация изображения), а также для передачи стиля очень важно, чтобы в процессе повышения разрешения изображения до исходного в каждый слой передавалась пространственная информация, теряемая в ходе понижения разрешения. Именно для этого и нужны пропускающие связи. Они позволяют сети смешивать абстрактную информацию высокого уровня, выделенную во время процесса понижения разрешения (то есть *стиль* изображения), с конкретной пространственной информацией, поступающей из предыдущих слоев сети (то есть *содержимое* изображения).

Пропускающие связи добавляются с помощью слоев еще одного типа: `Concatenate`.

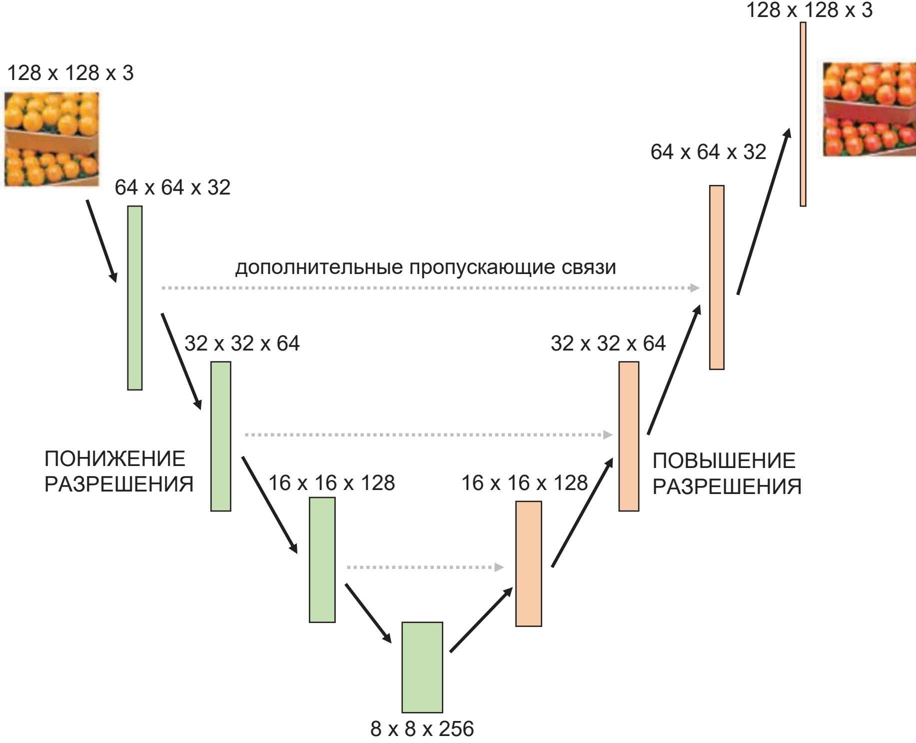



**Рис. 5.6.** Архитектура U-Net

### СЛОЙ CONCATENATE

Слой `Concatenate` просто соединяет наборы слоев вдоль определенной оси (по умолчанию последняя ось). Например, вот как в Keras можно объединить два предыдущих слоя, `x` и `y`:

```
Concatenate()([x,y])
```

В сети U-Net слои `Concatenate` используются для добавления связи между слоями в ветвях повышения и понижения разрешения. Слои соединяются вместе по размеру каналов, поэтому количество каналов увеличивается от $k$ до $2k$, а количество пространственных измерений остается неизменным.

Обратите внимание, что в слое `Concatenate` нет весов, которые должны корректироваться в процессе обучения; они просто «склеивают» предыдущие слои вместе.

### СЛОЙ НОРМАЛИЗАЦИИ ЭКЗЕМПЛЯРА

Генератор сети CycleGAN использует слои `InstanceNormalization` вместо `BatchNormalization`, что в задачах переноса стиля позволяет получить более удовлетворительные результаты.[1] Слой `InstanceNormalization` нормализует каждое наблюдение отдельно, а не весь пакет. В отличие от `BatchNormalization`, он не требует, чтобы параметры `mu` и `sigma` вычислялись как скользящее среднее во время обучения, потому что во время тестирования слой может нормализоваться для каждого экземпляра, как и во время обучения. Средние значения и стандартные отклонения, используемые для нормализации каждого слоя, рассчитываются для каждого канала и для каждого наблюдения.

Кроме того, слои `InstanceNormalization` в этой сети не имеют весов, которые требуется обучить, потому что мы не используем параметры масштабирования (`gamma`) и сдвига (`beta`).

[1] Дмитрий Ульянов (Dmitry Ulyanov), Андреа Ведальди (Andrea Vedaldi) и Виктор Лемпицкий (Victor Lempitsky), «Instance Normalization: The Missing Ingredient for Fast Stylization», 27 июля 2016, https://arxiv.org/pdf/1607.08022.pdf.

На рис. 5.7 показана разница между пакетной нормализацией и нормализацией экземпляра, а также между двумя другими методами нормализации (нормализация слоя и группы).

Здесь N — ось пакетов, C — ось каналов, а (H, W) представляют пространственные оси. Куб представляет входной тензор для слоя нормализации. Пикселы синего цвета нормализуются по одному и тому же среднему значению и дисперсии (рассчитываются по значениям этих пикселов).

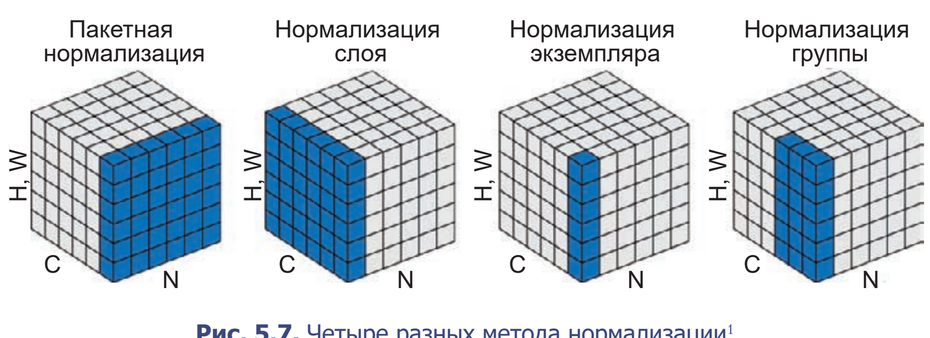



**Рис. 5.7.** Четыре разных метода нормализации[1]

Отметим, что генератор содержит и один новый слой — `InstanceNormalization`.

Располагая всем необходимым, можно с помощью Keras создать генератор U-Net (см. листинг 5.2).

**Листинг 5.2.** Генератор U-Net

```
def build_generator_unet(self):

    def downsample(layer_input, filters, f_size=4):
        d = Conv2D(filters, kernel_size=f_size
            , strides=2, padding='same')(layer_input)
        d = InstanceNormalization(axis = -1, center = False, scale = False)(d)
        d = Activation('relu')(d)
        return d
```

[1] Источник: Юйсинь Ву (Yuxin Wu) и Каймин Хе (Kaiming He), «Group Normalization», 22 марта 2018, https://arxiv.org/pdf/1803.08494.pdf.

```
def upsample(layer_input, skip_input, filters, f_size=4, dropout_rate=0):
    u = UpSampling2D(size=2)(layer_input)
    u = Conv2D(filters, kernel_size=f_size, strides=1, padding='same')(u)
    u = InstanceNormalization(axis = -1, center = False, scale = False)(u)
    u = Activation('relu')(u)
    if dropout_rate:
        u = Dropout(dropout_rate)(u)

    u = Concatenate()([u, skip_input])
    return u

# Входное изображение
img = Input(shape=self.img_shape)

# Понижение разрешения ❶
d1 = downsample(img, self.gen_n_filters)
d2 = downsample(d1, self.gen_n_filters*2)
d3 = downsample(d2, self.gen_n_filters*4)
d4 = downsample(d3, self.gen_n_filters*8)

# Повышение разрешения ❷
u1 = upsample(d4, d3, self.gen_n_filters*4)
u2 = upsample(u1, d2, self.gen_n_filters*2)
u3 = upsample(u2, d1, self.gen_n_filters)

u4 = UpSampling2D(size=2)(u3)

output = Conv2D(self.channels, kernel_size=4, strides=1
       , padding='same', activation='tanh')(u4)

return Model(img, output)
```

❶ Генератор состоит из двух половинок. Первая уменьшает разрешение изображения, используя слои `Conv2D` с шагом 2.

❷ Вторая повышает разрешение, возвращая тензор того же размера, что и исходное изображение. Блоки, увеличивающие разрешение, содержат слои `Concatenate`, придающие сети архитектуру U-Net.

## Дискриминаторы

Дискриминаторы, рассматривавшиеся ранее, возвращали единственное число: оценку вероятности того, что входное изображение является «реальным». Дискриминаторы в CycleGAN, которые мы построим, будут возвращать одноканальный тензор 8 × 8. Это обусловлено тем, что CycleGAN наследует архитектуру дискриминатора от модели, известной как *PatchGAN*, где дискриминатор делит изображение на квадратные фрагменты («за-

платки» — patches) с перекрытием и оценивает реальность фрагментов, а не всего изображения целиком. Поэтому-то на выходе дискриминатор возвращает тензор, содержащий вероятность для каждого фрагмента, а не единственное число.

Обратите внимание, вероятности для фрагментов предсказываются одновременно: передавая изображение в сеть, мы не делим его на фрагменты вручную. Деление происходит естественным образом, как следствие сверточной архитектуры дискриминатора. Преимущество использования дискриминатора PatchGAN в том, что функция потерь может измерить, насколько хорошо дискриминатор различает *стиль* изображений, а не их *содержимое*. Так как каждый отдельный элемент в предсказании дискриминатора основан на небольшом фрагменте изображения, он вынужден оценивать стиль фрагмента, а не его содержимое. Это именно то, что нам нужно; наш дискриминатор должен хорошо различать изображения, отличающиеся стилем, а не содержимым.

В листинге 5.3 показан код, создающий дискриминаторы с помощью Keras.

**Листинг 5.3.** Дискриминаторы

```
def build_discriminator(self):

    def conv4(layer_input,filters, stride = 2, norm=True):
        y = Conv2D(filters, kernel_size=4, strides=stride
               , padding='same')(layer_input)

        if norm:
            y = InstanceNormalization(axis = -1, center = False,
                                          scale = False)(y)

        y = LeakyReLU(0.2)(y)

        return y

    img = Input(shape=self.img_shape)

    y = conv4(img, self.disc_n_filters, stride = 2, norm = False) ❶
    y = conv4(y, self.disc_n_filters*2, stride = 2)
    y = conv4(y, self.disc_n_filters*4, stride = 2)
    y = conv4(y, self.disc_n_filters*8, stride = 1)

    output = Conv2D(1, kernel_size=4, strides=1, padding='same')(y) ❷

    return Model(img, output)
```

❶ Дискриминатор CycleGAN — это последовательность сверточных слоев с нормализацией экземпляра (кроме первого слоя).

❷ Последний слой — это сверточный слой с одним фильтром и без активации.

## Компиляция CycleGAN

Напомним, наша цель — создать набор моделей, способных преобразовать изображение из области *A* (например, изображение яблок) в область *B* (например, изображение апельсинов) и наоборот. Значит, мы должны скомпилировать четыре разные модели, два генератора и два дискриминатора:

`g_AB`

: Учится преобразовывать изображения из области *A* в область *B*.

`g_BA`

: Учится преобразовывать изображения из области *B* в область *A*.

`d_A`

: Учится отличать реальные изображения из области *A* от созданных генератором `g_BA`.

`d_B`

: Учится отличать реальные изображения из области *B* от созданных генератором `g_AB`.

Мы можем скомпилировать два дискриминатора напрямую, как показано в листинге 5.4, потому что у нас есть входные данные (изображения из каждой области) и выходные (бинарные ответы: 1, если изображение принадлежит области, или 0, если изображение сгенерировано искусственно).

**Листинг 5.4.** Компиляция дискриминатора

```
self.d_A = self.build_discriminator()
self.d_B = self.build_discriminator()
self.d_A.compile(loss='mse',
    optimizer=Adam(self.learning_rate, 0.5),
    metrics=['accuracy'])
```

```
self.d_B.compile(loss='mse',
    optimizer=Adam(self.learning_rate, 0.5),
    metrics=['accuracy'])
```

Однако скомпилировать генераторы напрямую нельзя из-за отсутствия парных изображений в наборе данных. Вместо этого оценим генераторы одновременно по трем критериям:

1. *Правдоподобие*. Признаются ли реальными изображения, созданные генератором, соответствующим дискриминатором? Например, результат генератора `g_BA` признается правдоподобным дискриминатором `d_A`, а результат `g_AB` признается правдоподобным дискриминатором `d_B`.
2. *Обратимость*. Если применить два генератора друг за другом (в обоих направлениях), получится ли исходное изображение? Отметим, что сеть CycleGAN получила свое название от критерия *циклической* обратимости.
3. *Идентичность*. Если применить каждый генератор к изображениям из своей целевой области, то останется ли изображение неизменным?

В листинге 5.5 показано, как скомпилировать модель, стремящуюся соблюсти эти три критерия (числовые маркеры в коде соответствуют пунктам списка выше).

**Листинг 5.5.** Конструирование комбинированной модели для обучения генераторов

```
self.g_AB = self.build_generator_unet()
self.g_BA = self.build_generator_unet()

self.d_A.trainable = False
self.d_B.trainable = False

img_A = Input(shape=self.img_shape)
img_B = Input(shape=self.img_shape)
fake_A = self.g_BA(img_B)
fake_B = self.g_AB(img_A)

valid_A = self.d_A(fake_A)
valid_B = self.d_B(fake_B) ❶

reconstr_A = self.g_BA(fake_B)
reconstr_B = self.g_AB(fake_A) ❷

img_A_id = self.g_BA(img_A)
img_B_id = self.g_AB(img_B) ❸
```

```
self.combined = Model(inputs=[img_A, img_B],
                        outputs=[ valid_A, valid_B,
                                reconstr_A, reconstr_B,
                                img_A_id, img_B_id ])

self.combined.compile(loss=['mse', 'mse',
                            'mae', 'mae',
                            'mae', 'mae'],
                    loss_weights=[
                        self.lambda_validation
                        , self.lambda_validation
                        , self.lambda_reconstr
                        , self.lambda_reconstr
                        , self.lambda_id
                        , self.lambda_id
                    ],
                    optimizer=optimizer)
```

Комбинированная модель принимает пакет изображений из каждой области и для каждой возвращает три выходных значения (оценки соответствия трем критериям) — всего шесть выходных значений. Подчеркнем: мы фиксируем весовые коэффициенты дискриминатора, что типично для генеративно-состязательных сетей, потому что комбинированная модель обучает только генераторы, несмотря на то что дискриминатор тоже участвует в ее работе. Общая сумма потерь взвешивается суммой потерь для каждого критерия. Для оценки критерия правдоподобия используется среднеквадратичная ошибка — результат дискриминатора сравнивается с ответом 1 (реальное) или 0 (поддельное), а для других двух критериев (обратимость и идентичность) используется средняя абсолютная ошибка.

## Обучение CycleGAN

Скомпилировав дискриминаторы и комбинированную модель, можно приступить к обучению. Этот подход соответствует стандартной практике использования генеративно-состязательных сетей, когда обучение дискриминаторов чередуется с обучением генераторов (листинг 5.6).

**Листинг 5.6.** Обучение CycleGAN

```
batch_size = 1
patch = int(self.img_rows / 2**4)
self.disc_patch = (patch, patch, 1)

valid = np.ones((batch_size,) + self.disc_patch) ❶
fake = np.zeros((batch_size,) + self.disc_patch)
```

```
for epoch in range(self.epoch, epochs):
    for batch_i, (imgs_A, imgs_B) in enumerate(data_loader.load_batch(batch_
                                                    size)):

        fake_B = self.g_AB.predict(imgs_A) ❷
        fake_A = self.g_BA.predict(imgs_B)

        dA_loss_real = self.d_A.train_on_batch(imgs_A, valid)
        dA_loss_fake = self.d_A.train_on_batch(fake_A, fake)
        dA_loss = 0.5 * np.add(dA_loss_real, dA_loss_fake)

        dB_loss_real = self.d_B.train_on_batch(imgs_B, valid)
        dB_loss_fake = self.d_B.train_on_batch(fake_B, fake)
        dB_loss = 0.5 * np.add(dB_loss_real, dB_loss_fake)
        d_loss = 0.5 * np.add(dA_loss, dB_loss)

        g_loss = self.combined.train_on_batch([imgs_A, imgs_B],
                                               [valid, valid,
                                               imgs_A, imgs_B,
                                               imgs_A, imgs_B])
```

❶ Мы используем ответ 1 для реальных изображений и 0 для сгенерированных. Для каждого фрагмента определяется свой, отдельный ответ, потому что используется дискриминатор PatchGAN.

❷ Для обучения дискриминаторов сначала с помощью генераторов создаются пакеты поддельных изображений, а затем и реальные, и поддельные изображения используются для обучения обоих дискриминаторов. Как правило, при обучении CycleGAN размер пакета выбирается равным 1 (одно изображение).

❸ Генераторы обучаются комбинированной моделью вместе за один шаг. Шесть выходов соответствуют шести функциям потерь, которые мы определили выше, на этапе компиляции.

## Анализ CycleGAN

Теперь посмотрим, как CycleGAN справляется с нашим простым набором изображений яблок и апельсинов и как изменение весовых параметров в функции потерь может повлиять на результаты. Мы уже видели на рис. 5.3 пример выходных данных, полученных моделью CycleGAN. Теперь, получив представление об архитектуре CycleGAN, вы без труда определите, что это изображение представляет три критерия, по которым оценивается объединенная модель: правдоподобие, обратимость и идентичность. Добавим подписи в это изображение, соответствующие функциям, чтобы можно было

увидеть это более четко (рис. 5.8). Обучение сети прошло успешно, потому что каждый генератор визуально изменяет входное изображение и делает его похожим на реальное изображение из противоположной области. Кроме того, когда генераторы применяются друг за другом, разница между входным и восстановленным изображением получается минимальной. Наконец, когда каждый генератор применяется к изображению из его собственной области, изображение почти не меняется.

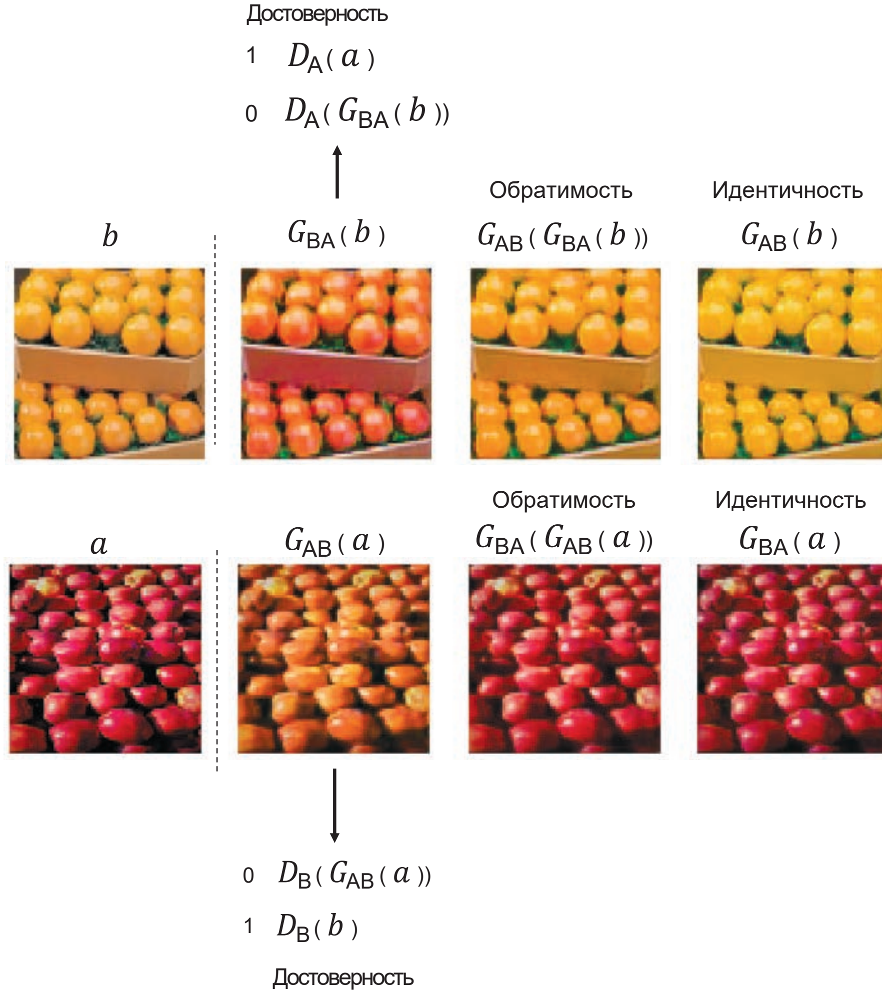



**Рис. 5.8.** Результаты, полученные комбинированной моделью и используемые для вычисления общей потери

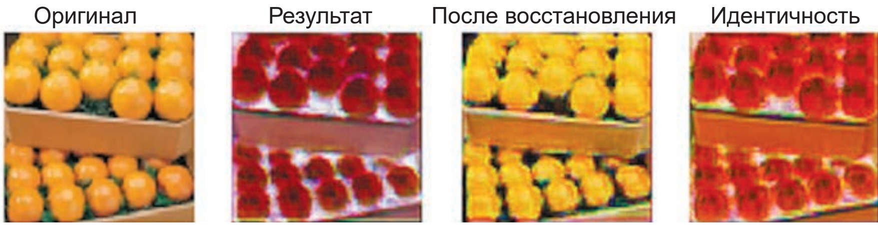



**Рис. 5.9.** Результаты, полученные сетью CycleGAN, после установки веса идентичности равным нулю

В статье с описанием CycleGAN потеря идентичности была включена как дополнение к обязательным потерям обратимости и правдоподобия. Чтобы показать важность члена в функции потерь, отвечающего за идентичность, посмотрим, что получится, если удалить его, установив соответствующий весовой параметр равным нулю (рис. 5.9).

Сети CycleGAN удалось превратить апельсины в яблоки, но цвет лотка изменился с черного на белый из-за отсутствия члена потери идентичности, предотвращающего изменение цвета фона. Член идентичности помогает отрегулировать генератор, чтобы тот корректировал только те части изображения, которые должны быть преобразованы, и не более того.

Эксперимент подчеркивает важность взвешивания трех функций потерь. Если потеря идентичности получит слишком маленький вес, то возникнет проблема смещения цвета. Если же потере идентичности придать слишком большой вес, то CycleGAN будет меньше стремиться изменить исходное изображение, чтобы оно выглядело как изображение из противоположной области.

## CycleGAN, рисующая в стиле Моне

Познакомившись с базовой структурой CycleGAN, обратим внимание на наиболее интересные и впечатляющие применения этой техники. Так, в статье с описанием CycleGAN упомянуто одно из выдающихся достижений — возможность научить модель превращать имеющиеся фотографии в картины, нарисованные в стиле конкретного художника. Кроме того, благодаря своей симметричной природе, модель CycleGAN способна выполнять обратное преобразование — превращать картины художника в реалистичные фотографии. Загрузим набор данных monet2photo:

```
bash ./scripts/download_cyclegan_data.sh monet2photo
```

Теперь сконструируем модель с параметрами так, как показано в листинге 5.7:

**Листинг 5.7.** Определение сети CycleGAN для рисования в стиле Моне

```
gan = CycleGAN(
        input_dim = (256,256,3)
        , learning_rate = 0.0002
        , lambda_validation = 1
        , lambda_reconstr = 10
        , lambda_id = 5
        , generator_type = 'resnet'
        , gen_n_filters = 32
        , disc_n_filters = 64
        )
```

## Генераторы (ResNet)

Теперь познакомимся с новым типом архитектуры генератора — *остаточной сетью* (residual network), или ResNet[1]. Архитектура ResNet подобно U-Net позволяет информации перетекать от предыдущих слоев к последующим. Однако вместо создания U-образной формы путем соединения слоев из части, понижающей разрешение, с соответствующими слоями в части, повышающей разрешение, ResNet строится из остаточных блоков, накладываемых стопкой друг на друга, где каждый блок имеет пропускающую связь, соединяющую вход и выход блока перед передачей результата следующему слою. Структура одного остаточного блока показана на рис. 5.10.

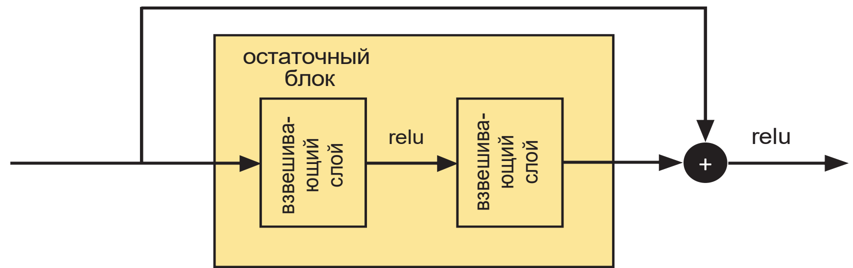



**Рис. 5.10.** Один остаточный блок

[1] Каймин Хе (Kaiming He) и др., «Deep Residual Learning for Image Recognition», 10 декабря 2015, https://arxiv.org/abs/1512.03385.

Взвешивающие слои в нашей сети CycleGAN — это сверточные слои с нормализацией экземпляра. В Keras остаточный блок можно определить так, как показано в листинге 5.8.

**Листинг 5.8.** Определение остаточного блока в Keras

```
from keras.layers.merge import add

def residual(layer_input, filters):
    shortcut = layer_input
    y = Conv2D(filters, kernel_size=(3, 3), strides=1, padding='same')
                                                       (layer_input)
    y = InstanceNormalization(axis = -1, center = False, scale = False)(y)
    y = Activation('relu')(y)
    y = Conv2D(filters, kernel_size=(3, 3), strides=1, padding='same')(y)
    y = InstanceNormalization(axis = -1, center = False, scale = False)(y)

    return add([shortcut, y])
```

По обе стороны от остаточных блоков в генераторе ResNet также присутствуют слои, понижающие и повышающие разрешение. Общая архитектура ResNet показана на рис. 5.11.

На практике архитектуры ResNet могут содержать до нескольких сотен и даже тысяч слоев и не страдать проблемой *затухания градиента*, когда градиенты на ранних слоях оказываются очень маленькими, из-за чего скорость обучения оказывается очень низкой. Это связано с тем, что градиенты ошибок могут свободно распространяться по сети через пропускающие связи, которые являются частью остаточных блоков. Кроме того, считается, что добавление дополнительных слоев не приводит к снижению точности

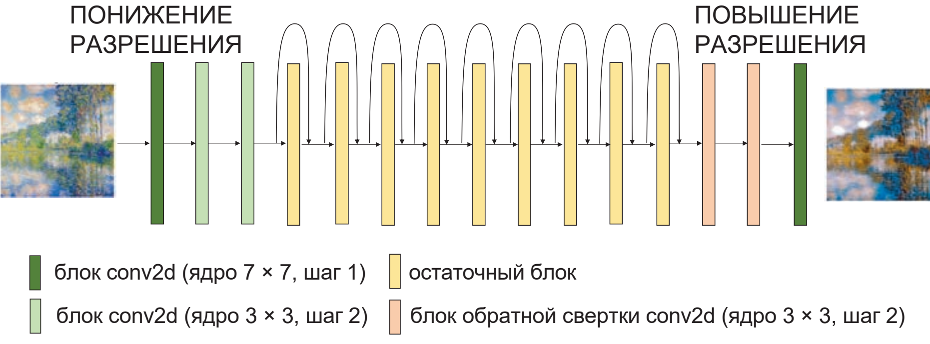



**Рис. 5.11.** Генератор ResNet

модели, потому что пропускающие связи гарантируют свободное перемещение информации об идентичности из предыдущего слоя, если никакие дополнительные информативные признаки не удается извлечь.

## Анализ CycleGAN

В оригинальной статье с описанием CycleGAN модель обучалась на протяжении 200 эпох, чтобы достичь выдающихся результатов по переносу стиля художника на фотографии. На рис. 5.12 показаны изображения, созданные каждым генератором на разных этапах обучения, чтобы продемонстрировать прогресс в обучении модели превращать картины Моне в фотографии и наоборот. В верхнем ряду мы видим, как отличительные цвета и мазки, используемые Моне, преобразуются в более естественные цвета и плавные переходы, характерные для фотографий. Аналогичным образом, в нижнем ряду демонстрируется обратный процесс — обучение генератора преобразованию фотографии в картину, которую мог бы написать Моне.

На рис. 5.13 показаны некоторые результаты из оригинальной статьи, которых удалось достичь модели после обучения на протяжении 200 эпох.

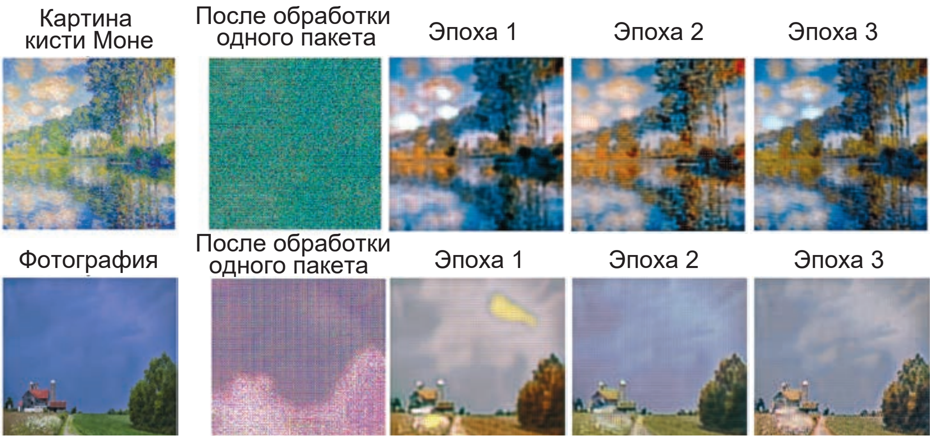



**Рис. 5.12.** Результаты, получаемые на разных этапах обучения

Картина кисти Моне | Сгенерированная фотография

Оригинальная фотография | Сгенерированная картина в стиле Моне

**Рис. 5.13.** Результаты, достигнутые после обучения на протяжении 200 эпох[1]

## Нейронный перенос стиля

Выше мы увидели, как CycleGAN может преобразовывать изображения между двумя областями, когда изображения в обучающем наборе могут не иметь соответствующих им парных изображений. Теперь рассмотрим другое применение техники передачи стиля — без использования обучающего набора, когда просто требуется перенести стиль одного изображения на другое, как показано на рис. 5.14. Этот прием известен как *нейронный перенос стиля*[2].

[1] Источник: Чжу (Zhu) и др., 2017, https://junyanz.github.io/CycleGAN.

[2] Леон А. Гатис (Leon A. Gatys), Александр С. Экер (Alexander S. Ecker) и Матиас Бетге (Matthias Bethge), «A Neural Algorithm of Artistic Style», 26 августа 2015, https://arxiv.org/abs/1508.06576.

**Рис. 5.14.** Пример нейронного переноса стиля[1]

Идея основана на предпосылке минимизации функции потерь — взвешенной суммы трех различных компонентов:

*Потеря содержимого*

Нам нужно, чтобы комбинированное изображение имело то же содержимое, что и базовое.

*Потеря стиля*

Нам нужно, чтобы комбинированное изображение имело тот же общий стиль, что и изображение стиля.

*Потеря общей дисперсии*

Нам нужно, чтобы комбинированное изображение не выглядело мозаичным.

Минимизируем эту потерю с помощью градиентного спуска, обновляя значение каждого пиксела на величину, пропорциональную отрицательному градиенту функции потерь, на протяжении многих итераций. При таком подходе потеря будет постепенно уменьшаться с каждой итерацией, и мы получим изображение, объединяющее содержимое одного изображения со стилем другого.

Оптимизация сгенерированного результата с помощью градиентного спуска отличается от привычного нам способа решения задач генеративного

[1] Источник: Гатис (Gatys) и др., 2015, https://arxiv.org/abs/1508.06576.

моделирования. Ранее мы обучали глубокие нейронные сети VAE или GAN методом обратного распространения ошибки по всей сети, чтобы извлечь информацию из обучающего набора и обобщить ее для создания новых изображений. Здесь мы не можем использовать этот способ, потому что у нас имеется только два изображения: базовое изображение и изображение стиля. Однако, как будет показано далее, мы все еще можем использовать предварительно обученную глубокую нейронную сеть для предоставления важной информации о каждом изображении в функциях потери. Начнем с определения трех отдельных функций потерь, являющихся основой механизма нейронного переноса стиля.

## Потеря содержимого

Потеря содержимого измеряет степень отличия двух изображений с точки зрения основного содержимого и его размещения. Два изображения с похожими сценами (например, две фотографии с рядом одних и тех же зданий, снятых при разном освещении и/или под разным углом) должны иметь меньшую потерю, чем два изображения с совершенно разными сценами. Простого сравнения пикселов двух изображений недостаточно, потому что пикселы даже в двух разных изображениях с одной и той же сценой будут иметь разные значения. Нам совсем не нужно, чтобы функция потери содержимого учитывала значения отдельных пикселов; она должна оценивать изображения по наличию и приблизительному положению высокоуровневых объектов вроде *здания*, *небо* или *река*.

Мы уже встречались с этой идеей. Основная предпосылка глубокого обучения — нейронная сеть, обученная распознаванию содержимого изображения, естественным образом выявляет высокоуровневые признаки в более глубоких слоях, комбинируя более простые признаки, выявленные предыдущими слоями. То есть нам нужна глубокая нейронная сеть, которая уже успешно обучена распознаванию содержимого изображений, чтобы можно было внедриться в наиболее глубокий ее слой и извлечь высокоуровневые признаки данного входного изображения. Измерив среднеквадратичную ошибку между результатом для базового и текущего комбинированного изображения, мы получим нашу функцию потери содержимого!

Далее используем предварительно обученную сеть VGG19. Это 19-слойная сверточная нейронная сеть, обученная для классификации изображений в тысячу категорий объектов на более чем миллионе изображений из набора данных ImageNet (рис. 5.15).

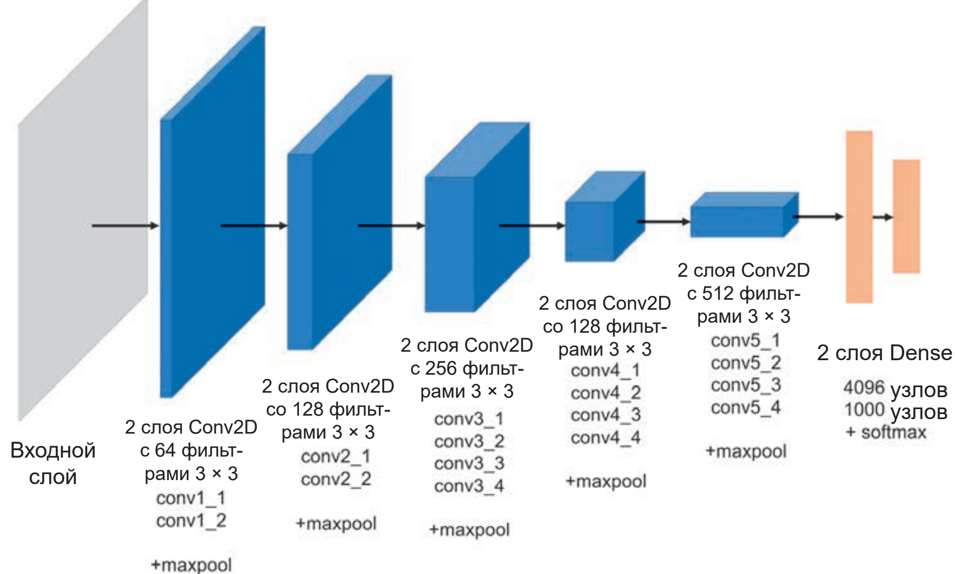



**Рис. 5.15.** Модель VGG19

В листинге 5.9 представлен фрагмент кода, который вычисляет величину потери содержимого между двумя изображениями, заимствованный из примера нейронной передачи стиля в официальном репозитории Keras (http://bit.ly/2FlVU0P). Чтобы опробовать пример и поэкспериментировать с параметрами, советуем использовать этот репозиторий в качестве отправной точки.

**Листинг 5.9.** Функция потери содержимого

```
from keras.applications import vgg19 ❶
from keras import backend as K

base_image_path = '/path_to_images/base_image.jpg'
style_reference_image_path = '/path_to_images/styled_image.jpg'

content_weight = 0.01

base_image = K.variable(preprocess_image(base_image_path)) ❷
style_reference_image = K.variable(preprocess_image(style_reference_image_
                                                     path))
combination_image = K.placeholder((1, img_nrows, img_ncols, 3))

input_tensor = K.concatenate([base_image,
```

```
                                   style_reference_image,
                                   combination_image], axis=0) ❸

model = vgg19.VGG19(input_tensor=input_tensor,
                    weights='imagenet', include_top=False) ❹

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
layer_features = outputs_dict['block5_conv2'] ❺

base_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :] ❻

def content_loss(content, gen):
    return K.sum(K.square(gen – content))

content_loss = content_weight * content_loss(base_image_features
                                            , combination_features) ❼
```

❶ В состав библиотеки Keras входит предварительно обученная модель VGG19, которую можно импортировать.

❷ Мы определяем две переменные для хранения базового изображения и изображения стиля, а также переменную для хранения сгенерированного комбинированного изображения.

❸ Входной тензор для модели VGG19, объединяющий все три изображения.

❹ Здесь мы создаем экземпляр модели VGG19 и указываем входной тензор и начальные веса для загрузки. Параметр `include_top = False` указывает, что веса не должны загружаться в конечные полносвязанные слои сети, осуществляющие классификацию изображения, потому что нас интересуют только предшествующие им сверточные слои, фиксирующие высокоуровневые признаки входного изображения, а не фактические вероятности, которые исходная модель была обучена выводить.

❺ Для вычисления потери содержимого используем второй сверточный слой пятого блока. Выбор слоя влияет на то, как функция потерь будет определять «содержимое» и, следовательно, на свойства сгенерированного комбинированного изображения.

❻ Здесь мы извлекаем признаки базового и комбинированного изображений из входного тензора, пропущенного через сеть VGG19.

❼ Потеря содержимого определяется как сумма квадратов расстояний между выходами выбранного слоя для обоих изображений, умноженная на весовой параметр.

## Потеря стиля

Оценить потерю стиля немного сложнее, поскольку не вполне ясно, как измерить сходство стилей двух изображений. Решение, предложенное в оригинальной статье с описанием нейронного переноса стиля, основано на идее сходства шаблонов корреляции между картами признаков в данном слое для изображений со схожим стилем. Эту идею можно объяснить на следующем примере. Предположим, в сети VGG19 есть какой-то слой, один канал которого научился идентифицировать части изображения, окрашенные в зеленый цвет, другой научился определять наличие колючек (шипов), а третий — идентифицировать части изображения, окрашенные в коричневый цвет. Выходы каналов (карты признаков) для трех входов показаны на рис. 5.16.

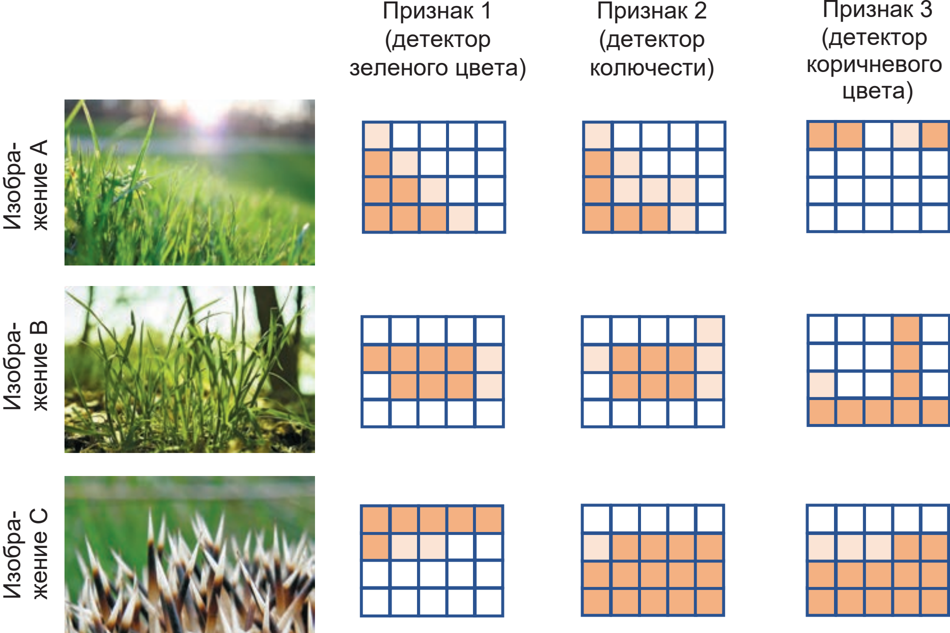



**Рис. 5.16.** Выходы трех каналов (карты признаков) для трех заданных входных изображений — более темные цвета соответствуют большим значениям

Как видим, изображения *A* и *B* похожи по стилю — оба они *травянистые*. Стиль изображения *C* немного отличается от стиля изображений *A* и *B*. Если посмотреть на карты признаков, можно заметить, что каналы *зеленого цвета* и *колючести* часто сильно коррелируются в одной и той же пространственной области на изображениях *A* и *B*, но не на изображении *C*. Напротив, каналы *коричневого цвета* и *колючести* часто коррелируют в одной и той же области на изображении *C*, но не на изображениях *A* и *B*. Чтобы измерить корреляцию двух карт признаков, можно сгладить их и вычислить скалярное произведение[1]. Большое значение указывает на значительное коррелирование карты признаков, малое — напротив, на то, что карты признаков не коррелируют.

Мы можем определить матрицу, содержащую скалярные произведения всех возможных пар признаков в слое. Эта матрица называется *матрицей Грама* (Gram matrix). На рис. 5.17 показаны матрицы Грама для трех признаков каждого из изображений.

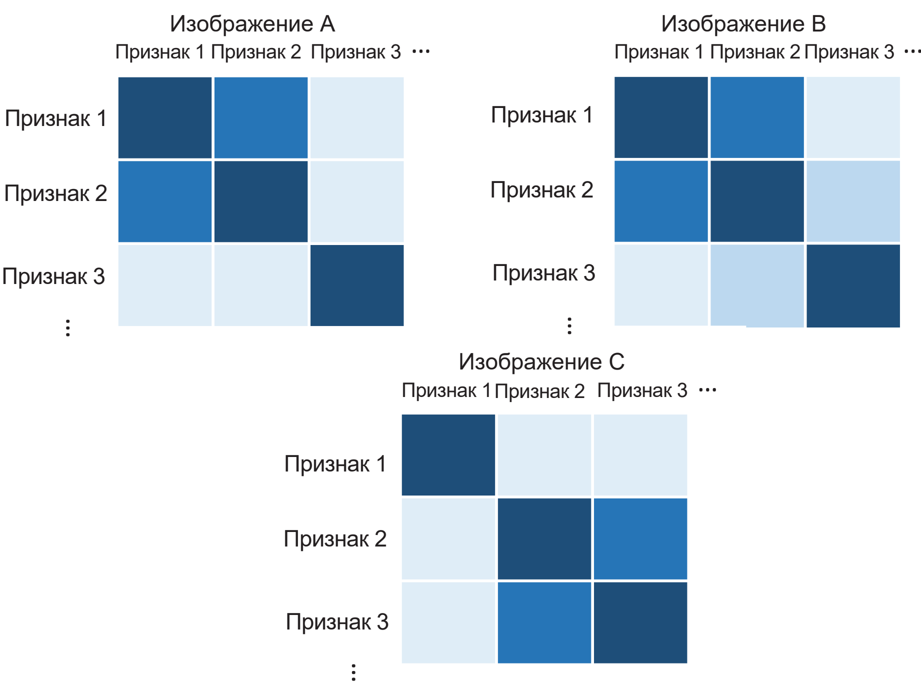



**Рис. 5.17.** Фрагменты матриц Грама для трех изображений — более темные цвета соответствуют большим значениям

[1] Чтобы вычислить скалярное произведение двух векторов, нужно попарно умножить их элементы и сложить результаты.

Как видим, изображения $A$ и $B$, похожие по стилю, имеют похожие матрицы Грама для выбранного слоя. Содержимое изображений может сильно отличаться, но матрицы Грама, отражающие корреляции между всеми парами признаков в слое, похожи.

Итак, чтобы вычислить потерю стиля, нужно вычислить матрицу Грама ($GM$) во всех слоях сети для базового и комбинированного изображений, сравнив их сходство с помощью суммы квадратов ошибок. Алгебраически потерю стиля между базовым ($S$) и сгенерированным ($G$) изображениями в данном слое ($l$) с размером $M_l$ (высота × ширина) и $N_l$ каналами можно записать следующим образом:

$$L_{GM}(S,G,l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} \left( GM[l](S)_{ij} - GM[l](G)_{ij} \right)^2.$$

Обратите внимание на то, как производится масштабирование, чтобы учесть количество каналов ($N_l$) и размер слоя ($M_l$). Это необходимо, поскольку общая потеря стиля вычисляется как взвешенная сумма для нескольких слоев, каждый из которых имеет разные размеры. Общая потеря стиля вычисляется следующим образом:

$$L_{style}(S,G) = \sum_{l=0}^{L} w_l L_{GM}(S,G,l).$$

С использованием библиотеки Keras вычисление потери стиля можно реализовать, как показано в листинге 5.10.[1]

**Листинг 5.10.** Функция потери стиля

```
style_loss = 0.0

def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return K.sum(K.square(S — C)) / (4.0 * (channels ** 2) * (size ** 2))
```

[1] Источник: GitHub, http://bit.ly/2FlVU0P.

```
feature_layers = ['block1_conv1', 'block2_conv1',
                  'block3_conv1', 'block4_conv1',
                  'block5_conv1'] ❶

for layer_name in feature_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :] ❷
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    style_loss += (style_weight / len(feature_layers)) * sl ❸
```

❶ Потеря стиля вычисляется по пяти слоям — первым сверточным слоям в каждом из пяти блоков модели VGG19.

❷ Здесь из входного тензора, пропускаемого через сеть VGG19, извлекаются признаки изображения стиля и комбинированного изображения.

❸ Потеря стиля масштабируется весовым параметром и числом слоев, по которым производились вычисления.

## Потеря общей дисперсии

Потеря общей дисперсии является просто мерой шума в комбинированном изображении. Чтобы судить о величине шума в изображении, можно сдвинуть его на один пиксел вправо и вычислить сумму квадратов разностей между смещенным и исходным изображениями. Для баланса можно выполнить ту же процедуру, сдвинув изображение на один пиксел вниз. Сумма этих двух значений представляет потерю общей дисперсии. В листинге 5.11 показано, как эта потеря вычисляется с помощью Keras.[1]

**Листинг 5.11.** Функция потери общей дисперсии

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_nrows – 1, :img_ncols – 1, :] – x[:, 1:, :img_ncols – 1, :]) ❶
    b = K.square(
        x[:, :img_nrows – 1, :img_ncols – 1, :] – x[:, :img_nrows – 1, 1:, :]) ❷
    return K.sum(K.pow(a + b, 1.25))

tv_loss = total_variation_weight * total_variation_loss(combination_image) ❸

loss = content_loss + style_loss + tv_loss ❹
```

[1] Источник: GitHub, http://bit.ly/2FlVU0P.

❶ Сумма квадратов разностей между исходным изображением и тем же изображением, сдвинутым на один пиксел вниз.

❷ Сумма квадратов разностей между исходным изображением и тем же изображением, сдвинутым на один пиксел вправо.

❸ Потеря общей дисперсии масштабируется весовым параметром.

❹ Полная потеря — это сумма потерь содержимого, стиля и общей дисперсии.

## Запуск нейронного переноса стиля

Процесс обучения включает градиентный спуск для минимизации функции потерь, описанной выше, в отношении пикселов в комбинированном изображении. Полный код — в сценарии neural_style_transfer.py (http://bit.ly/2FlVU0P), размещенном в официальном репозитории Keras. В листинге 5.12 представлен фрагмент кода, реализующий цикл обучения.

**Листинг 5.12.** Цикл обучения модели нейронного переноса стиля

```
from scipy.optimize import fmin_l_bfgs_b

iterations = 1000
x = preprocess_image(base_image_path) ❶

for i in range(iterations):
    x, min_val, info = fmin_l_bfgs_b( ❷
        evaluator.loss ❸
        , x.flatten()
        , fprime=evaluator.grads ❸
        , maxfun=20
        )
```

❶ В начале процесса обучения в роли *комбинированного* выступает исходное изображение.

❷ В каждой итерации текущее комбинированное изображение (в виде простого вектора) передается в функцию оптимизации `fmin_l_bfgs_b` из пакета `scipy.optimize`, которая выполняет один шаг градиентного спуска в соответствии с алгоритмом L-BFGS-B.

❸ `evaluator` — это объект, содержащий методы, которые вычисляют полную потерю, как описано выше, и градиенты потери относительно входного изображения.

## Анализ модели нейронного переноса стиля

На рис. 5.18 показан результат нейронной передачи стиля на трех разных этапах процесса обучения со следующими параметрами:

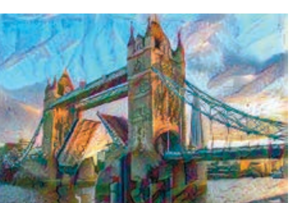
- `content_weight: 1`
- `style_weight: 100`
- `total_variation_weight: 20`

Как видим, с каждым шагом обучения алгоритм стилистически все больше приближается к стилевому изображению, теряя детали и сохраняя общую структуру базового изображения. Эта архитектура предлагает богатые возможности для экспериментов. Можно попробовать изменить весовые параметры в функции потерь или поменять слой, используемый для определения сходства содержимого, чтобы увидеть, как это повлияет на комбинированное изображение и скорость обучения. Также можно попытаться уменьшить вес, присваиваемый каждому слою в функции потери стиля, сместив модель в сторону передачи более тонких или более грубых элементов стиля.

Базовое изображение

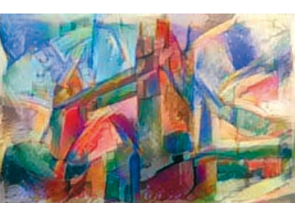
Изображение стиля

Итерация 1



Потеря: $9.25 \times 10^{10}$

Итерация 200



Потеря: $1.54 \times 10^{10}$

Итерация 400

Потеря: $1.44 \times 10^{10}$

**Рис. 5.18.** Результат нейронной передачи стиля после 1, 200 и 400 итераций

## Итоги

Итак, мы исследовали два разных способа создания новых художественных произведений.

Способ на основе CycleGAN позволяет создать модель, способную изучить общий стиль художника и переносить ее на фотографии, возвращая изображения, выглядящие так, будто они нарисованы художником. Эта модель позволяет выполнять и обратное преобразование: превращать картины в реалистичные фотографии. Важно отметить, что модели CycleGAN не требуются парные изображения из разных областей, и это делает ее чрезвычайно мощным и гибким инструментом.

Способ нейронной передачи стиля позволяет перенести стиль уникального изображения на базовое изображение с использованием обоснованно выбранной функции потерь, штрафующей модель за слишком большое отклонение от содержимого основного изображения и художественного стиля стилевого изображения и одновременно сохраняющей гладкость изображения на выходе. Этот способ коммерциализирован многими известными приложениями, реализующими смешивание фотографий пользователя с заданным набором стилистических рисунков.

Теперь перенесемся из области генеративного моделирования изображений в область генеративного моделирования текста, где перед нами откроются новые проблемы.

ГЛАВА 6

# Литературное творчество

В этой главе рассмотрены методы конструирования генеративных моделей для текстовых данных. Текстовые данные и изображения имеют несколько ключевых различий, из-за чего многие методы, прекрасно работающие с изображениями, плохо подходят для работы с текстовыми данными. В частности:

- Текстовые данные состоят из дискретных фрагментов (символов или слов), тогда как пикселы на изображении являются точками в непрерывном цветовом спектре. Мы легко можем сделать зеленый пиксел синим, но превратить слово *кошка* в слово *собака* намного сложнее. Из этого следует, что мы легко можем применить прием обратного распространения к изображениям, вычисляя градиент функции потерь в отношении отдельных пикселов и определяя направление, в котором должны изменяться цвета пикселов для минимизации потери. Применить обратное распространение к дискретным текстовым данным обычным способом не получится, поэтому нужно найти способ обойти эту проблему.

- Текстовые данные имеют временно́е измерение, но не имеют пространственного измерения, тогда как изображения имеют два пространственных измерения, но не имеют временно́го измерения. Порядок слов в текстовых данных очень важен, и, к примеру, слова, расположенные в обратном порядке, теряют смысл, тогда как изображения обычно могут переворачиваться без потери смыслового наполнения. Кроме того, в последовательностях слов нередко существуют протяженные зависимости, которые модель должна уметь выявлять, например: ответ на вопрос или перенос вперед контекста местоимения. В случае с изображениями все пикселы могут обрабатываться одновременно.

- Текстовые данные очень чувствительны к небольшим изменениям в отдельных единицах (словах или символах). Изображения, как правило, менее чувствительны к изменениям в отдельных пиксельных единицах — в изображении дома все равно можно узнать дом, даже изменив некоторые пикселы, формирующие его изображение. Однако изменение даже одного слова в тексте может радикально изменить его смысл или полностью лишить смысла. Это очень затрудняет обучение модели созданию связного текста, так как каждое слово жизненно важно для общего значения отрывка.
- Текстовые данные имеют структуру, основанную на грамматических правилах, тогда как изображения не следуют каким-либо правилам, регламентирующим, как должны назначаться значения пикселам. Например, в любом контексте было бы грамматически безграмотно написать: «Кошка сидит на обладающий». Существуют также семантические правила, которые чрезвычайно трудно смоделировать; например, выражение «я в пляже» семантически построено неправильно, хотя грамматически в нем нет ничего плохого.

В моделировании текста достигнут большой прогресс, но подходы к решению вышеупомянутых проблем все еще остаются предметом исследований. Рассмотрим далее одну из наиболее устоявшихся моделей, широко используемых для генерации последовательных данных, таких как текст, — рекуррентную нейронную сеть (Recurrent Neural Network, RNN), и в частности слой долгой краткосрочной памяти (Long Short-Term Memory, LSTM). А затем познакомимся с некоторыми новыми методами, показавшими многообещающие результаты в области генерации пар вопрос/ответ. Но сначала бросим взгляд на историю о тюрьме, заключенные которой организовали литературное общество.

## Литературное общество для проблемных правонарушителей

Тюремный надзиратель Эдвард Сопп ненавидел свою работу. Он проводил дни, наблюдая за заключенными, почти не имея свободного времени, чтобы предаться своей истинной страсти — написанию коротких рассказов. В конце концов вдохновение покинуло Соппа и ему понадобился способ, позволяющий восстановить способность к сочинительству.

Однажды ему в голову пришла блестящая идея, воплотив которую он смог бы создавать новые художественные произведения в своем стиле, и в то же время занять заключенных. Идея была в том, чтобы заставить заключенных писать рассказы за него! Он основал новое общество LSTM (Literary Society for Troublesome Miscreants — литературное общество для проблемных правонарушителей)[1].

Тюрьма, где работает Эдвард, — необычная, потому что в ней имеется только одна большая камера, где содержатся 256 заключенных. У каждого заключенного есть свое мнение о том, как следует продолжить текущий рассказ Эдварда. Каждый день Эдвард сообщает заключенным последнее слово из своего рассказа, а те должны скорректировать свое представление о текущем состоянии рассказа, опираясь на новое слово и вчерашние мнения заключенных.

Каждый заключенный корректирует свое мнение, сопоставляя новое слово и мнения других заключенных со своим собственным мнением в прошлом. Прежде всего, каждый решает, что из вчерашнего мнения он должен забыть, опираясь на новое слово и мнения других заключенных в камере. Затем все заключенные формируют новые мысли и решают, какие из них следует добавить к прежним мнениям, которые решено перенести из предыдущего дня. После этого каждый формирует свое новое мнение на текущий день.

Однако заключенные скрытны и не всегда делятся своими мыслями с сокамерниками. Они также индивидуально используют последнее выбранное слово и мнения других заключенных, чтобы решить, какую часть своего мнения раскрыть.

Когда Эдвард хочет получить следующее слово, он просит заключенных сообщить свои мнения охраннику, стоящему у двери камеры, который объединяет полученную информацию и выбирает, какое слово будет добавлено в конец рассказа. Это новое слово затем сообщается заключенным в камере, и процесс продолжается, пока рассказ не будет завершен.

Чтобы обучить заключенных и охранника, Эдвард передает написанные им короткие последовательности слов в камеру и проверяет, насколько правильно заключенные выбрали следующее слово. Он сообщает им, насколь-

[1] Здесь игра слов: аббревиатура LSTM в мире машинного обучения расшифровывается как «Long Short-Term Memory» (долгая краткосрочная память). — *Примеч. пер.*

ко правильный выбор они сделали, и постепенно заключенные обучаются писать рассказы в своем собственном уникальном стиле.

После множества итераций обучения Эдвард обнаружил, что система достигла достаточно высокого уровня совершенства в создании реалистичного текста. Правда, текстам не хватает семантической структуры, но они, безусловно, демонстрируют характеристики, свойственные его предыдущим рассказам. Наконец, удовлетворенный результатами Сопп публикует сборник рассказов под названием «Басни Э. Соппа».

## Сети с долгой краткосрочной памятью

История мистера Соппа и его басен, написанных коллективом заключенных, весьма напоминает одну из наиболее успешных и широко используемых методик глубокого обучения для создания таких последовательных данных, как текст: сеть с долгой краткосрочной памятью (Long Short-Term Memory, LSTM). Сеть LSTM – это разновидность *рекуррентной нейронной сети* (Recurrent Neural Network, RNN). Сети RNN содержат рекуррентный слой (или *ячейку*), способный обрабатывать последовательные данные и генерировать новые данные, характерные для определенного временного шага, формируя часть входных данных для следующего временного шага, благодаря чему информация из прошлого может влиять на прогноз на текущем временном шаге. Таким образом, под *сетью LSTM* подразумевается нейронная сеть с рекуррентным слоем LSTM.

Когда сети RNN только появились, рекуррентные слои были очень простыми и состояли исключительно из оператора `tanh`, который гарантировал масштабирование информации, передаваемой между временными шагами, в диапазон от –1 до 1. Но позднее выяснилось, что это решение страдает проблемой затухания градиента и плохо подходит для создания длинных последовательностей.

Впервые ячейки LSTM упомянуты в 1997 году в статье[1] Зеппа Хохрайтера и Юргена Шмидхубера. В этой статье авторы описывают, почему LSTM не страдают проблемой затухания градиента, характерной для обычных RNN, и могут обучаться на последовательностях длиной в сотни временных шагов. С тех пор архитектура LSTM адаптирована и улучшена. В настоящее

---

[1] Sepp Hochreiter и Jürgen Schmidhuber, Long Short-Term Memory», *Neural Computation* 9 (1997): 1735–1780, http://bit.ly/2In7NnH.

время ее вариации широко используются и доступны в виде слоев Keras как *управляемые рекуррентные блоки* (Gated Recurrent Unit, GRU). Итак, посмотрим, как с помощью Keras построить простую сеть LSTM, способную генерировать текст в стиле басен Эзопа.

## Ваша первая сеть LSTM

Сначала нужно получить исходные данные.

В качестве основы используем коллекцию басен Эзопа (http://bit.ly/2QQEf5T), скачав ее с сайта Project Gutenberg (http://www.gutenberg.net/). Напомним, это коллекция бесплатных электронных книг в виде простых текстовых файлов. Она является отличным источником данных, которые можно использовать для формирования текстовых моделей глубокого обучения.

Загрузим данные:

```
bash ./scripts/download_gutenburg_data.sh 11339 aesop
```

Теперь рассмотрим шаги, которые нужно выполнить, чтобы привести данные в форму, пригодную для обучения сети LSTM. Соответствующий код вы найдете в блокноте 06_01_lstm_text_train.ipynb.

### Лексемизация

Первый шаг: очистка и лексемизация текста. *Лексемизация* — это процесс разделения текста на отдельные единицы, лексемы, такие как слова или символы.

Порядок лексемизации текста зависит от целей, стоящих перед моделью. Деление текста на слова и на символы имеет свои плюсы и минусы, и выбор, который мы сделаем, будет влиять на то, как следует очистить текст перед моделированием и что получится на выходе модели.

При лексемизации по словам весь текст можно преобразовать в нижний регистр, чтобы гарантировать одинаковую лексемизацию слов с заглавными буквами в начале предложений и тех же слов в середине предложений. Однако иногда это может быть нежелательно: например, сохранение заглавных букв в некоторых именах собственных (имена людей, географические названия) может дать определенные выгоды, потому что они будут лексемизироваться независимо.

- Текстовый *словарь* (набор отдельных слов в обучающем наборе) может получиться очень большим, при этом некоторые слова будут использоваться в тексте очень редко или даже только один раз. Такие «уникальные» слова может быть целесообразно заменить специальной лексемой, обозначающей *неизвестное слово*, а не включать их как отдельные лексемы, чтобы уменьшить количество весов в нейронной сети.
- Слова можно *упростить*, сведя их к простейшей форме, с тем чтобы глаголы в разном времени лексемизировались в одну лексему. Например: *просмотр*, *просмотреть*, *просмотрел* и *просмотрели* — все эти слова будут преобразованы в лексему *просмотр*.
- Знаки препинания нужно будет тоже лексемизировать или вообще отбросить.
- Применение лексемизации слов означает, что модель никогда не сможет сгенерировать слово, не вошедшее в учебный словарь.

Если выбрать лексемизацию по символам, то модель сможет генерировать последовательности символов, образующие новые слова, не вошедшие в учебный словарь. Это может быть желательно в одних контекстах, но нежелательно в других.

- Заглавные буквы могут преобразовываться в строчные или сохраняться в виде отдельных лексем.
- Словарь при использовании лексемизации по символам обычно получается куда менее объемным. Это способствует увеличению скорости обучения модели, так как в конечном выходном слое содержится меньше весов.

В этом примере мы используем лексемизацию по словам с преобразованием символов в нижний регистр и без упрощения (*стемминга*) слов. Мы также будем лексемизировать знаки препинания, чтобы модель смогла предсказывать, когда, например, должны заканчиваться предложения или открываться/закрываться прямая речь. Наконец, мы заменим последовательности символов перевода строки, отделяющие рассказы, блоком символов *новый рассказ*, ||||||||||||||||||. Благодаря этому, генерируя текст с помощью модели, мы сможем передать ей этот блок символов, уведомляющий о том, что модель должна начать новый рассказ.

Код в листинге 6.1 очищает текст и выполняет лексемизацию.

**Листинг 6.1.** Лексемизация

```
import re
from keras.preprocessing.text import Tokenizer

filename = "./data/aesop/data.txt"

with open(filename, encoding='utf-8-sig') as f:
    text = f.read()

seq_length = 20
start_story = '| ' * seq_length

# ОЧИСТКА
text = text.lower()
text = start_story + text
text = text.replace('\n\n\n\n\n', start_story)
text = text.replace('\n', ' ')
text = re.sub(' +', '. ', text).strip()
text = text.replace('..', '.')

text = re.sub('([!»#$ %&()*+,-./:;<=>?@[\]^_`{|}~])', r' \1 ', text)
text = re.sub('\s{2,}', ' ', text)

# ЛЕКСЕМИЗАЦИЯ
tokenizer = Tokenizer(char_level = False, filters = '')
tokenizer.fit_on_texts([text])
total_words = len(tokenizer.word_index) + 1
token_list = tokenizer.texts_to_sequences([text])[0]
```

На рис. 6.1 показан фрагмент текста после очистки.

На рис. 6.2 показан словарь лексем с соответствующими им индексами, а также фрагмент лексемизированного текста с соответствующими словами, выделенными зеленым цветом.

' | | | | | | | | | | | | | | | | | | | | the fox and the grapes . a hungry fox saw some fine bunches of grapes hanging from a vine that was trained along a high trellis , and did his best to reach them by jumping as high as he could into the air . but it was all in vain , for they were just out of reach : so he gave up trying , and walked away with an air of dignity and unconcern , remarking , " i thought those grapes were ripe , but i see now they are quite sour . " | | | | | | | | | | | | | | | | | | | | the goose that laid the golden eggs . a man and his wife had the good fortune to possess a goose which laid a golden egg every day . lucky though they were , they soon began to think they were not getting rich fast enough , and , imagining the bird must be made of gold inside , they decided to kill it in order to secure the whole store of precious metal at once . but when they cut it open they found it was just like any other goose . thus , they neither got rich all at once , as they had hoped , nor enjoyed any longer the daily addition to their wealth . much wants more and loses all . | | | | | | | | | | | | | | | | | | | | the cat and the mice . there was once a house that was overrun with mice . a cat heard of this , and said to herself , " that\'s the place for me , " and off she went and took up her quarters in the house , and caught the mice one by one and ate them . at last the mice could stand it no longer , and they determined to take to their holes and stay there . " that\'s awkward , " said the cat to herself : " the only thing to do is to coax them out by a trick . " so she considered a while , and then climbed up the wall and let herself hang down by her hind legs from a peg , and pretended to be dead . by and by

**Рис. 6.1.** Текст после очистки

```
tokenizer.word_index

{'|': 1,
 ',': 2,
 'the': 3,
 'and': 4,
 '.': 5,
 'a': 6,
 'to': 7,
 '"': 8,
 'of': 9,
 'he': 10,
 'his': 11,
 'was': 12,
```

```
token_list

1,    |
3,    the
56,   fox
4,    and
3,    the
940,  grapes
5,    .
6,    a
382,  hungry
56,   fox
94,   saw
```

**Рис. 6.2.** Словарь, отображающий слова в индексы (слева), и текст после лексемизации (справа)

## Создание набора данных

Наша сеть LSTM будет обучаться предсказывать следующее слово по имеющейся последовательности, с учетом слов, предшествующих текущей позиции. Например, мы можем передать в модель текст «жадный кот и» и ожидать, что модель выведет подходящее следующее слово (например, «собака», а не «в»). Длина последовательности, используемой для обучения модели, является параметром процесса обучения. Для этого примера мы решили использовать последовательность длиной в 20 слов, поэтому разобьем текст на фрагменты по 20 слов. Всего может быть построено 50 416 таких последовательностей, поэтому наш набор обучающих данных `X` представляет собой массив с формой `[50416, 20]`. В ответ на каждую последовательность модель будет возвращать следующее за ней слово, представленное в виде вектора прямого кодирования с длиной 4169 (количество отдельных слов в словаре). То есть ответ является массивом двоичных цифр с формой `[50416, 4169]`.

Создание набора данных осуществляет код, представленный в листинге 6.2.

**Листинг 6.2.** Создание набора данных

```
import numpy as np
from keras.utils import np_utils
```

```
def generate_sequences(token_list, step):

    X = []
    y = []

    for i in range(0, len(token_list) - seq_length, step):
        X.append(token_list[i: i + seq_length])
        y.append(token_list[i + seq_length])

    y = np_utils.to_categorical(y, num_classes = total_words)

    num_seq = len(X)
    print('Number of sequences:', num_seq, "\n")

    return X, y, num_seq

step = 1
seq_length = 20
X, y, num_seq = generate_sequences(token_list, step)

X = np.array(X)
y = np.array(y)
```

## Архитектура модели LSTM

На рис. 6.3 показана общая архитектура модели. На вход модели подается последовательность целочисленных индексов лексем, а на выходе возвращается вероятность для каждого слова в словаре, указывающая, что именно это слово должно быть очередным в последовательности. Чтобы понять, как работает модель, нужно познакомиться с двумя новыми типами слоев: `Embedding` и `LSTM`.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None)              0
_________________________________________________________________
embedding_1 (Embedding)      (None, None, 100)         416900
_________________________________________________________________
lstm_1 (LSTM)                (None, 256)               365568
_________________________________________________________________
dense_1 (Dense)              (None, 4169)              1071433
=================================================================
Total params: 1,853,901
Trainable params: 1,853,901
Non-trainable params: 0
```

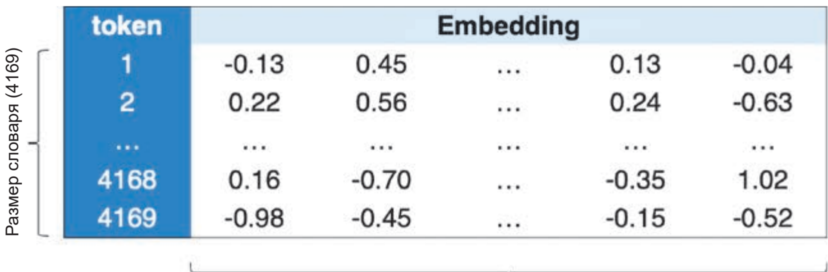
**Рис. 6.3.** Архитектура модели LSTM

## Слой Embedding

Слой `Embedding` (создания векторного представления) — это, по сути, таблица соответствий, преобразующая каждую лексему в вектор с длиной `embedding_size` (рис. 6.4). То есть количество весов в этом слое равно размеру словаря, умноженному на `embedding_size`.

Слой `Input` передает тензор целочисленных представлений лексем с формой `[batch_size, seq_length]` в слой `Embedding`, который возвращает тензор с формой `[batch_size, seq_length, embedding_size]`, который затем передается в слой `LSTM` (рис. 6.5).

Целочисленное представление каждой лексемы преобразуется в непрерывный вектор, позволяя модели выучить векторное представление каждого слова, которое может обновляться на этапе обратного распространения. Мы также могли бы использовать метод прямого кодирования для каждой входной лексемы, но подход на основе векторного представления с использованием слоя `Embedding` предпочтительнее, поскольку делает это векторное представление обучаемым, что увеличивает гибкость модели при принятии решения о том, какое представление должна иметь каждая лексема, чтобы повысить качество модели.



| token | Embedding | | | | |
|---|---|---|---|---|---|
| 1 | -0.13 | 0.45 | ... | 0.13 | -0.04 |
| 2 | 0.22 | 0.56 | ... | 0.24 | -0.63 |
| ... | ... | ... | ... | ... | ... |
| 4168 | 0.16 | -0.70 | ... | -0.35 | 1.02 |
| 4169 | -0.98 | -0.45 | ... | -0.15 | -0.52 |



**Рис. 6.4.** Слой Embedding — это таблица преобразования целочисленных представлений лексем в векторы

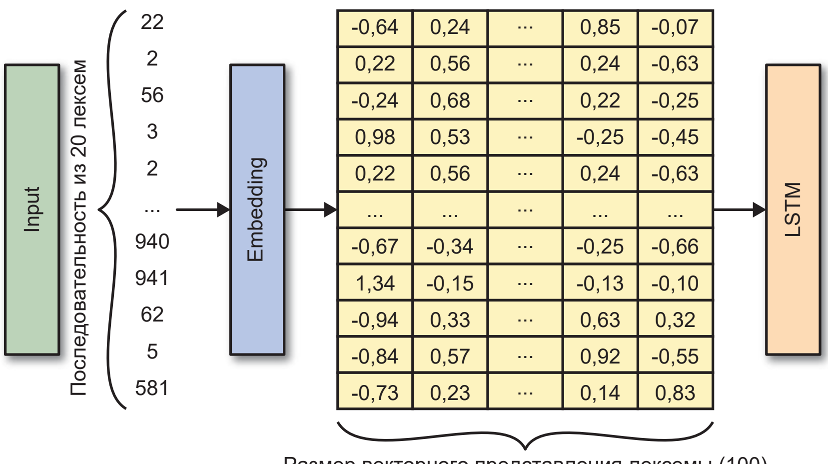



**Рис. 6.5.** Как единственная последовательность пересекает слой Embedding

## Слой LSTM

Чтобы понять, как работает слой LSTM, сначала познакомимся с работой рекуррентного слоя.

Рекуррентный слой обладает особой возможностью обрабатывать последовательные входные данные $[x_1, ..., x_n]$. Он состоит из ячейки, обновляющей свое *скрытое состояние* $h_t$ по мере прохождения каждого элемента последовательности $x_t$. Скрытое состояние — вектор, длина которого равна количеству *узлов* в ячейке, — можно рассматривать как текущее понимание последовательности ячейкой. На временном шаге $t$ ячейка использует предыдущее значение скрытого состояния $h_{t-1}$ с данными из текущего временного шага $x_t$, чтобы получить обновленный вектор скрытого состояния $h_t$. Этот рекуррентный процесс продолжается до конца последовательности. Когда последовательность закончится, слой возвращает окончательное скрытое состояние ячейки $h_n$, которое затем передается в следующий слой сети. Этот процесс показан на рис. 6.6.

Чтобы объяснить работу слоя более подробно, развернем процесс и посмотрим, как одиночная последовательность пересекает слой (рис. 6.7).

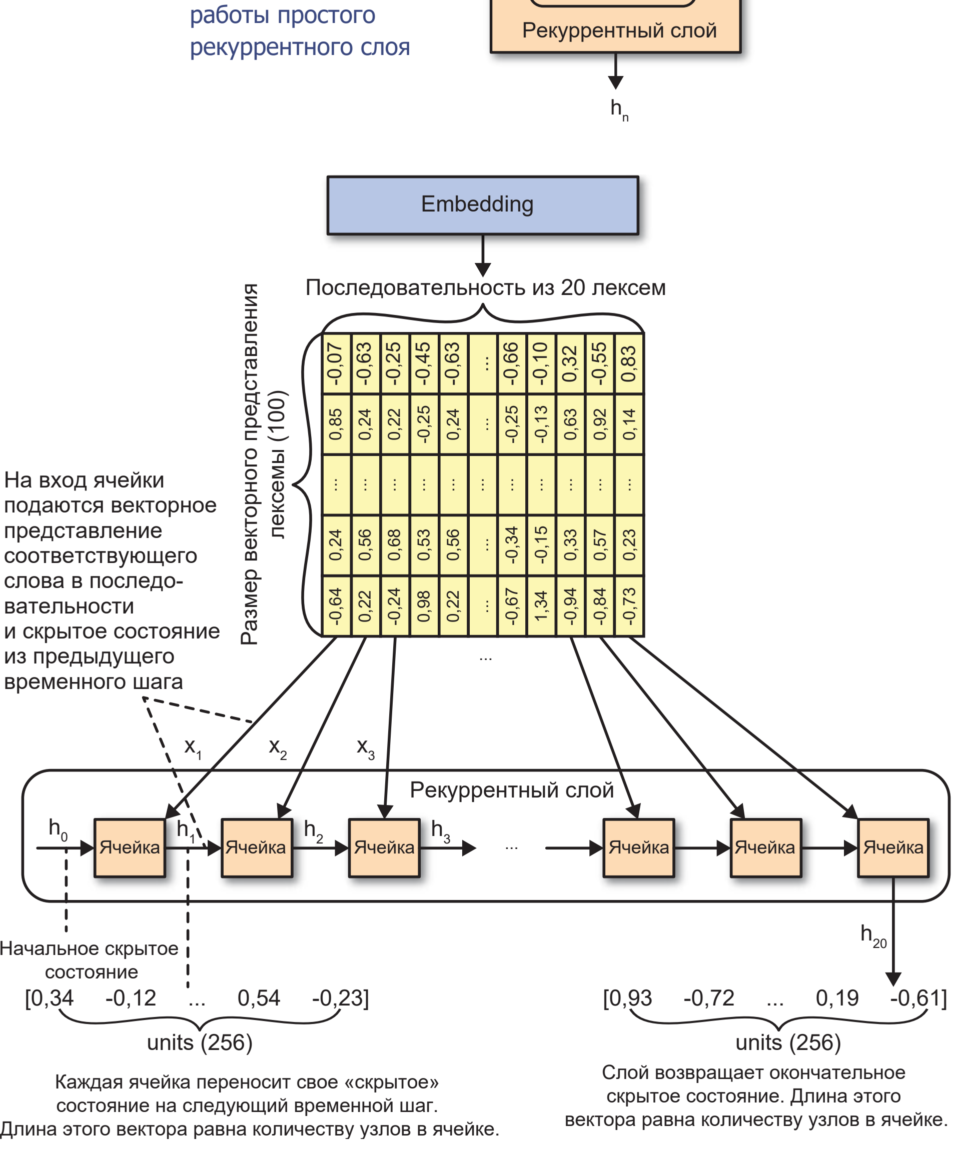



**Рис. 6.6.** Схема работы простого рекуррентного слоя

**Рис. 6.7.** Как одиночная последовательность пересекает рекуррентный слой

Эта диаграмма рекуррентного процесса, где каждая ячейка представляет отдельный временной шаг, наглядно иллюстрирует, как скрытое состояние постоянно обновляется при прохождении через ячейки. Здесь ясно видно, что предыдущее скрытое состояние смешивается с текущей точкой данных в последовательности (то есть с векторным представлением текущего слова) и создается следующее скрытое состояние. Результатом слоя является достигнутое скрытое состояние ячейки после обработки всех слов во входной последовательности. Важно помнить, что все ячейки на этой диаграмме имеют одинаковые веса (потому что на самом деле это одна и та же ячейка). Между этой диаграммой и рис. 6.6 нет никакой разницы; это просто другой способ представления механики рекуррентного слоя.

Выбор названия «скрытое состояние» для результата, получаемого ячейкой, является следствием неудачного соглашения. На самом деле оно не скрыто, и вы не должны считать его таковым. В действительности последнее скрытое состояние становится результатом всего слоя, и кроме того, далее в этой главе не раз будет показана возможность доступа к скрытому состоянию на каждом отдельном временном шаге.

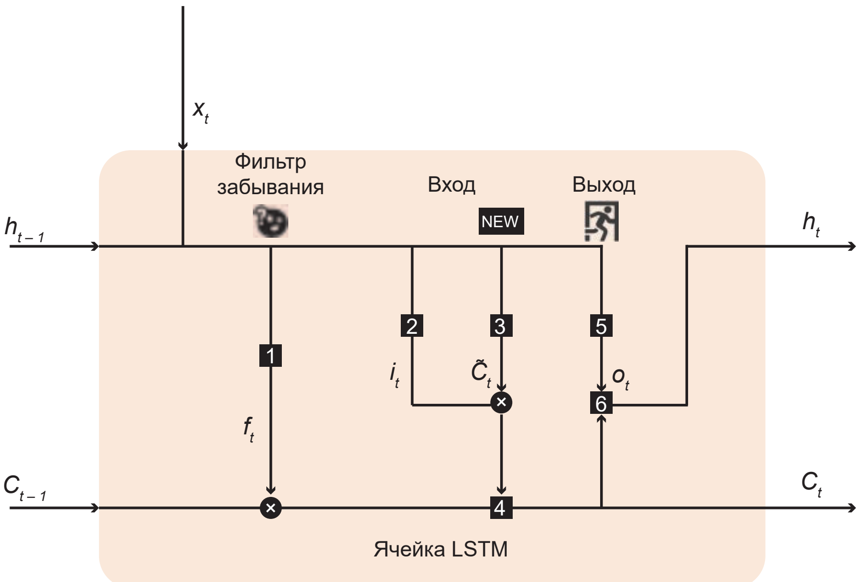
## Ячейка LSTM

Теперь, увидев, как работает простой рекуррентный слой, рассмотрим отдельную ячейку LSTM.

Задача ячейки LSTM — вернуть новое скрытое состояние $h_t$, опираясь на предыдущее скрытое состояние $h_{t-1}$ и векторное представление текущего слова $x_t$. Напомним, длина $h_t$ равна количеству узлов в LSTM. Это — параметр, устанавливаемый при определении слоя и не имеющий никакого отношения к длине последовательности. Не путайте понятия *ячейка* и *узел*. В слое LSTM есть одна ячейка, определяемая количеством содержащихся в ней узлов (так же, как в тюремной камере из нашей предыстории содержалось много заключенных). Мы часто изображаем рекуррентный слой в виде развернутой цепочки ячеек, поскольку это помогает наглядно представить, как происходит обновление скрытого состояния на каждом временном шаге.

Ячейка LSTM поддерживает состояние $C_t$, которое можно рассматривать как внутреннее ее убеждение о текущем состоянии последовательности. Это состояние отличается от скрытого состояния $h_t$, которое в конечном итоге воз-

вращается ячейкой после последнего временного шага. Состояние $C_t$ имеет ту же длину, что и скрытое состояние (равную количеству узлов в ячейке). Рассмотрим подробнее отдельную ячейку и то, как происходит обновление скрытого состояния (рис. 6.8).




$$\boxed{1}\quad f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

$$\boxed{2}\quad i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$

$$\boxed{3}\quad \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$\boxed{4}\quad C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$\boxed{5}\quad o_t = \sigma\left(W_o\ [h_{t-1}, x_t] + b_o\right)$$

$$\boxed{6}\quad h_t = o_t * \tanh\left(C_t\right)$$

**Рис. 6.8.** Ячейка LSTM

Скрытое состояние обновляется в шесть приемов:

1. Скрытое состояние, полученное на предыдущем временном шаге $h_{t-1}$, и векторное представление текущего слова $x_t$ объединяются и передаются через фильтр *забывания*, являющийся обычным полносвязанным слоем с матрицей весов $W_f$, смещением $b_f$ и функцией активации *sigmoid*. Вектор $f_t$ на выходе фильтра имеет длину, равную количеству узлов в ячейке, и содержит значения в диапазоне от 0 до 1, определяющие, какая доля предыдущего состояния ячейки $C_{t-1}$ должна сохраниться.

2. Объединенный вектор передается через входной фильтр, который, подобно фильтру забывания, является полносвязанным слоем с матрицей весов $W_i$, смещением $b_i$ и функцией активации *sigmoid*. Выходной вектор $i_t$ этого фильтра имеет длину, равную числу узлов в ячейке, и содержит значения в диапазоне от 0 до 1, определяющие, какая доля новой информации будет добавлена в предыдущее состояние ячейки $C_{t-1}$.

3. Объединенный вектор передается через полносвязанный слой с матрицей весов $W_C$, смещением $b_C$ и функцией активации *tanh*, чтобы получить вектор $\tilde{C}_t$, содержащий новую информацию, которую ячейка решила сохранить. Он содержит значения в диапазоне от $-1$ до 1, и его длина тоже равна количеству узлов в ячейке.

4. Векторы $f_t$ и $C_{t-1}$ умножаются поэлементно и складываются с поэлементным произведением $i_t$ и $\tilde{C}_t$. Это равноценно забыванию части предыдущего состояния ячейки и добавлению новой релевантной информации для получения обновленного состояния ячейки $C_t$.

5. Исходный объединенный вектор также передается через выходной фильтр: полносвязанный слой с матрицей весов $W_o$, смещением $b_o$ и активацией *sigmoid*. Вектор результата $o_t$ имеет длину, равную количеству узлов в ячейке, и хранит значения в диапазоне от 0 до 1, определяющие, какая доля обновленного состояния $C_t$ может покинуть ячейку.

6. $o_t$ умножаются поэлементно на обновленное состояние ячейки $C_t$, а затем применяется активация *tanh*, чтобы создать новое скрытое состояние $h_t$.

В листинге 6.3 показан код, конструирующий сеть LSTM.

**Листинг 6.3.** Конструирование сети LSTM

```
from keras.layers import Dense, LSTM, Input, Embedding, Dropout
from keras.models import Model
from keras.optimizers import RMSprop

n_units = 256
embedding_size = 100

text_in = Input(shape = (None,))
x = Embedding(total_words, embedding_size)(text_in)
x = LSTM(n_units)(x)
x = Dropout(0.2)(x)
text_out = Dense(total_words, activation = 'softmax')(x)

model = Model(text_in, text_out)

opti = RMSprop(lr = 0.001)
model.compile(loss='categorical_crossentropy', optimizer=opti)

epochs = 100
batch_size = 32
model.fit(X, y, epochs=epochs, batch_size=batch_size, shuffle = True)
```

## Генерирование нового текста

Теперь, скомпилировав и обучив сеть LSTM, можно начать использовать ее для генерации длинных строк текста:

1. Передать в сеть имеющуюся последовательность слов и попросить ее предсказать следующее за ней слово.

2. Добавить это слово в конец последовательности и повторить.

Сеть будет выводить набор, включающий вероятности для всех слов в словаре, опираясь на которые мы можем делать выбор. Таким образом, генерировать текст можно стохастически. Кроме того, в процесс выбора можно ввести температурный параметр, чтобы указать, насколько детерминированным должен быть этот процесс (листинг 6.4).

**Листинг 6.4.** Генерирование текста с помощью сети LSTM

```
def sample_with_temp(preds, temperature=1.0): ❶
    # вспомогательная функция выбора индекса в массиве вероятностей
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
```

```
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probs = np.random.multinomial(1, preds, 1)
    return np.argmax(probs)

def generate_text(seed_text, next_words, model, max_sequence_len, temp):
    output_text = seed_text
    seed_text = start_story + seed_text ❷

    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([seed_text])[0] ❸
        token_list = token_list[-max_sequence_len:] ❹
        token_list = np.reshape(token_list, (1, max_sequence_len))

        probs = model.predict(token_list, verbose=0)[0] ❺
        y_class = sample_with_temp(probs, temperature = temp) ❻

        output_word = tokenizer.index_word[y_class] if y_class > 0 else ''

        if output_word == "|": ❼
            break

        seed_text += output_word + ' ' ❽
        output_text += output_word + ' '

    return output_text
```

❶ Эта функция взвешивает логарифмы вероятностей с коэффициентом `temperature` перед повторным применением функции *softmax*. Чем ближе значение температуры к нулю, тем более детерминированным получается выбор (выше вероятность выбора слова с наибольшей вероятностью), а температура, равная 1, означает, что слова выбираются с вероятностью, выведенной моделью.

❷ Начальный текст — это последовательность слов, которую требуется передать модели, чтобы запустить процесс генерации. Последовательность (которая может быть пустой) начинается с блока символов, отмечающего начало рассказа (||||||||||||||||||||).

❸ Слова преобразуются в список лексем.

❹ Сохраняются только последние `max_sequence_len` лексем. Слой LSTM может принимать на входе последовательности любой длины, но чем длиннее последовательность, тем больше времени требуется, чтобы сгенерировать следующее слово, поэтому длину последовательности желательно ограничить.

❺ Модель возвращает вероятность каждого слова в словаре стать следующим словом в последовательности.

❻ Вероятности передаются функции `sample_with_temp`, которая делает выбор с учетом параметра `temperature`.

❼ Если слово на выходе является маркером начала рассказа, то процесс генерирования слов прекращается, потому что таким способом модель сообщает, что хочет закончить этот рассказ и начать следующий!

❽ В противном случае новое слово добавляется в исходный текст для следующей итерации.

Давайте посмотрим, что генерирует эта сеть с двумя разными значениями `temperature` (рис. 6.9).

Есть несколько моментов, которые стоит отметить в этих двух отрывках. Во-первых, оба стилистически похожи на басни из обучающего набора. Оба открываются знакомым высказыванием персонажей басен, а текст в кавычках очень напоминает диалог, причем сходство усиливается использованием личных местоимений, которые сопровождаются словом *said* (сказал).

**Temperature 0.2**

```
the ass and his lion . a certain man who had an ass and a stag at them in a good . " do , " said the other saying , "
you but the ass to be so . to come and you are in the way of ever see how you are , in the man of a lion . " the lion
replied , " you not very much to me , and i will be a eyes upon you , and , as that you were for the good of a on tha
t men will be had all the dog to be a lion . as the cock came , and the ground for it in a very much to wolf , " you
take my oh , if you will do not get me in any time the a very way of some day . " well again , which had very well on
ly a man , he turned up and said to the , " but i see what you see how i are day : but you not one of them to be not
well ? " . he saw what be a of master in the ass was said to him , " the other , you are a man of me , and i were so
. " the wolf was so much to the man and said to his master , " you , in the friend of you are ; and then he should be
running up they found for you said he had a way to make them . " as the very way , for it was that he could not see a
t the eagle and said to him , " i will be a lion and i have was one day , and you know that you go again , but come a
nd see what you are of the better . "
```

**Temperature 1.0**

```
the great man and the time . a sheep master fell on a head and had been in a other to get them . so the ass , caught
a young horse and was by me , and so he got himself into a horse than the lion of their put out ; and why , he found
a lion in a ass . you must go , he they we not only could not a bull by one day the tortoise in gratitude in a place
live of the very do wish you . so , as he could not the fox no one make the ass of the day , where in the well have w
ould all she had been in a good first , but the other dog and then a cock in a moment . i'm not a lion , who had " co
me off , that do you see that you find that the lion , who they fell on to be very much more good good strength , and
have to a mouse go . when the other who is same , who had no town mouse , and found a cat up out of a great much . "
the better feed in the water , that she came to a bird . the eagle never do so good a frog or and drink ; so my maste
r , saw that the master is was well and the ass at my a very good time . the ox turned too use for a way to have to h
erself : but if the will very fox came to let him one as one as it , and man as he should there well on by the place
and stood there . for the time the lamb ass in the ass , for his the lost these all , should found turned down . a us
e of the hard went to make it , " and the never soon can be of out of a ass . " the master , so not much be a eagle n
ever should be oh ! " presently , not to be a ass , who do not how were heavy : i'm not a do you , you get off in the
water , and not come to we are very take the just so ever even in my master .
```

**Рис. 6.9.** Два фрагмента текста, сгенерированных сетью LSTM с двумя разными значениями параметра temperature

Во-вторых, текст, сгенерированный с параметром `temperature = 0.2`, выглядит более последовательным и связным, чем текст, сгенерированный с параметром `temperature = 1.0`, потому что низкие значения влекут более детерминированный выбор.

И наконец, ясно видно, что описанный прием плохо подходит для создания рассказов длиной больше нескольких предложений, потому что сеть LSTM не понимает смыслового значения генерируемых слов. Чтобы сгенерировать отрывки, имеющие больше шансов получиться семантически правильными, можно создать текстовый генератор с участием человека, где модель выводит 10 слов с наивысшей вероятностью, а окончательный выбор из этого списка делает человек. Это похоже на функцию предсказания текста в вашем мобильном телефоне, когда вам предлагается выбор из нескольких слов, которые могут следовать за текстом, который вы уже ввели. Для демонстрации на рис. 6.10 показаны 10 слов с наибольшей вероятностью следования за различными последовательностями (не из обучающего набора).

Модель может генерировать список следующих наиболее вероятных слов в пределах контекста. Например, хотя модель ничего не знает о частях речи, таких как существительные, глаголы, прилагательные и предлоги, она обычно способна разделить слова на эти классы и использовать их грамматически правильно. Модель также предположила, что басня об орле, скорее всего, должна начинаться с артикля *an*, а не *a*[1].

Пример со знаками препинания на рис. 6.10 показывает, что модель чувствительна даже к незначительным изменениям во входной последовательности. В первом отрывке («the lion said,» — «лев сказал,») модель предположила, что далее с вероятностью 98 % должны следовать двойные кавычки, открывающие прямую речь, потому что подобные выражения часто предшествуют прямой речи. Однако при добавлении союза «and» (и) после запятой сеть определила, что прямая речь далее маловероятна, так как диалог, скорее всего, заменен повествовательным предложением.

---

[1] В английском языке артикль *a* используется перед словами, начинающимися с согласного звука, а артикль *an* — с гласного. Английское слово «eagle» (орел) начинается с гласного звука. — *Примеч. пер.*

Noun

```
the fox and the stag . there was a
21.3%   : time
19.1%   : lion
15.7%   : man
11.6%   : ass
8.6%    : good
4.3%    : fox
2.0%    : a
1.8%    : once
1.7%    : old
1.3%    : dogs
```

Verb

```
the fox and the snake . one day a fox
49.9%   : came
19.8%   : saw
8.0%    : went
6.0%    : ,
5.5%    : was
4.2%    : were
1.2%    : said
1.1%    : go
0.4%    : had
0.3%    : put
```

Preposition

```
the dog and the hare . a dog was lying
68.6%   : in
21.6%   : on
6.7%    : at
0.6%    : by
0.5%    : into
0.4%    : from
0.3%    : to
0.3%    : with
0.2%    : about
0.2%    : for
```

Adjective / Verb

```
the farmer and his sheep . a farmer was
17.2%   : unable
8.3%    : afraid
6.1%    : so
5.4%    : good
5.0%    : lying
4.6%    : sitting
4.2%    : who
2.4%    : by
2.3%    : going
2.3%    : caught
```

Article

```
the eagle and the sea .
89.4%   : an
4.1%    : the
1.5%    : just
1.2%    : thus
0.7%    : two
0.7%    : presently
0.6%    : a
0.5%    : there
0.4%    : jupiter
0.1%    : at
```

Punctuation

```
the lion said ,
98.0%   : "
1.1%    : and
0.4%    : that
0.1%    : the
0.1%    : much
0.0%    : you
0.0%    : do
0.0%    : time
0.0%    : as
0.0%    : a
```

```
the lion said , and
84.4%   : the
5.2%    : you
3.4%    : that
3.0%    : if
0.3%    : well
0.2%    : "
0.2%    : i
0.2%    : my
0.2%    : we
0.2%    : very
```

**Рис. 6.10.** Распределение вероятностей слов, которые могли бы продолжить разные последовательности

## Расширения RNN

В предыдущем разделе был представлен простой пример обучения сети LSTM генерированию текста в заданном стиле. В этом разделе мы рассмотрим несколько расширений этой идеи.

## Многослойные рекуррентные сети

Сеть, которую мы только что рассмотрели, содержала всего один слой LSTM, но аналогичным образом можно обучать сети со множеством слоев LSTM, чтобы извлечь из текста более глубокие особенности. Для этого нужно установить параметр `return_sequence` в первом слое LSTM в значение `True`. Это заставит слой выводить скрытое состояние из каждого временного шага, а не только последнего. Второй слой LSTM, следующий за ним, может использовать скрытые состояния из первого слоя в качестве своих входных данных. Этот процесс изображен на рис. 6.11, а общая архитектура модели показана на рис. 6.12.

Код, конструирующий многослойную сеть LSTM, показан в листинге 6.5.

**Листинг 6.5.** Конструирование многослойной сети LSTM

```
text_in = Input(shape = (None,))
    embedding = Embedding(total_words, embedding_size)
    x = embedding(text_in)
    x = LSTM(n_units, return_sequences = True)(x)
    x = LSTM(n_units)(x)
    x = Dropout(0.2)(x)
    text_out = Dense(total_words, activation = 'softmax')(x)

    model = Model(text_in, text_out)
```

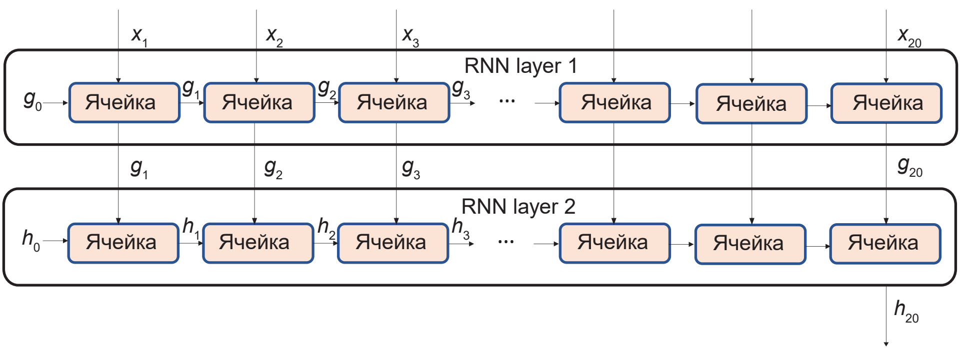



**Рис. 6.11.** Схема многослойной рекуррентной сети: $g_t$ обозначают скрытые состояния в первом слое, а $h_t$ — во втором слое

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None)              0
_________________________________________________________________
embedding_1 (Embedding)      (None, None, 100)         416900
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 256)         365568
_________________________________________________________________
lstm_2 (LSTM)                (None, 256)               525312
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 4169)              1071433
=================================================================
Total params: 2,379,213
Trainable params: 2,379,213
Non-trainable params: 0
_________________________________________________________________
```

**Рис. 6.12.** Многослойная сеть LSTM

## Управляемые рекуррентные блоки

Другой тип слоев, широко используемых в RNN, — *управляемые рекуррентные блоки* (Gated Recurrent Unit, GRU).[1] Вот основные его отличия от LSTM:

1. *Входной* фильтр и фильтр *забывания* замещаются фильтрами *сброса* и *обновления*.

2. Отсутствуют *состояние ячейки* и *выходной* фильтр, имеется только *скрытое состояние*, выводимое ячейкой.

Скрытое состояние обновляется в четыре приема, как показано на рис. 6.13.

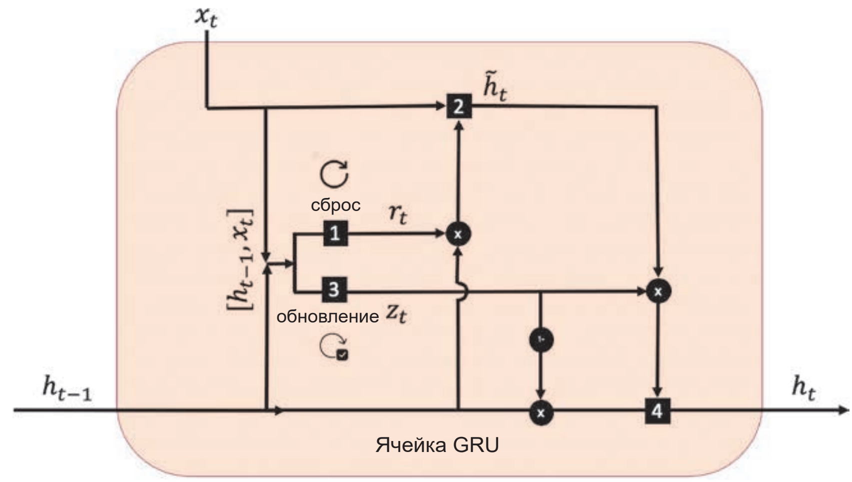
Вот как это происходит:

1. Скрытое состояние предыдущего временного шага $h_{t-1}$ и векторное представление текущего слова $x_t$ объединяются и используются для создания фильтра *сброса*. Этот фильтр организован как полносвязанный слой

[1] Кюнгнхён Чо (Kyunghyun Cho) и др., «Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation», 3 июня 2014, https://arxiv.org/abs/1406.1078.

с матрицей весов $W_r$ и функцией активации *sigmoid*. Вектор результата $r_t$ имеет длину, равную количеству узлов в ячейке, и хранит в диапазоне от 0 до 1 значения, которые определяют, какая доля предыдущего скрытого состояния $h_{t-1}$ должна использоваться для вычисления новых убеждений ячейки.

2. Фильтр сброса применяется к скрытому состоянию $h_{t-1}$, и результат объединяется с векторным представлением текущего слова $x_t$. Затем объединенный вектор передается в полносвязанный слой с матрицей весов $W$ и функцией активации *tanh*, а на выходе получается вектор $\tilde{h}_t$, в котором хранятся новые убеждения ячейки. Он имеет длину, равную количеству узлов в ячейке, и хранит значения от –1 до 1.




1 $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$

2 $\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$

3 $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$

4 $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$

**Рис. 6.13.** Единственная ячейка GRU

3. Результат объединения скрытого состояния предыдущего временного шага $h_{t-1}$ и векторного представления текущего слова $x_t$ также используется для создания фильтра *обновления*. Этот фильтр организован как полносвязанный слой с матрицей весов $W_z$ и функцией активации *sigmoid*. Вектор результата $z_t$ имеет длину, равную количеству узлов в ячейке, и хранит в диапазоне от 0 до 1 значения, которые определяют, какая доля новых убеждений $\tilde{h}_t$ будет смешана с текущим скрытым состоянием $h_{t-1}$.
4. Новые убеждения ячейки $\tilde{h}_t$ и текущее скрытое состояние $h_{t-1}$ смешиваются в пропорции, определяемой фильтром обновления $z_t$, после чего получается обновленное скрытое состояние $h_t$, выводимое из ячейки.

## Двунаправленные ячейки

Для задач прогнозирования, когда модели доступен весь текст для анализа, нет оснований обрабатывать последовательность только в прямом направлении — ее также можно обработать в обратном направлении. Воспользоваться этим преимуществом позволяет слой `Bidirectional`. Он хранит два набора скрытых состояний: одно является результатом обработки последовательности в прямом направлении, другое — в обратном. Таким образом, слой может обучаться на информации и предшествующей заданному временному шагу, и следующей за ним.

В Keras это реализуется как обертка вокруг рекуррентного слоя:

```
layer = Bidirectional(GRU(100))
```

Скрытые состояния в получившемся слое — это векторы с длиной, равной удвоенному количеству узлов в обернутой ячейке (объединение прямого и обратного скрытого состояния). То есть скрытые состояния слоя в примере — это векторы с длиной 200.

## Модели кодировщик-декодировщик

Итак, мы выяснили, как с помощью сетей LSTM можно генерировать продолжение существующей текстовой последовательности. Мы также увидели, как единственный слой LSTM может последовательно обрабатывать данные и обновлять скрытое состояние, отражающее текущее представление слоя о последовательности. Передавая окончательное скрытое состояние в полносвязанный слой, сеть может вывести распределение вероятности для следующего слова.

Однако в некоторых случаях целью является не предсказание единственного слова, продолжающего последовательность, а создание новой последовательности слов, каким-то образом связанной со входной последовательностью. Вот некоторые примеры подобных задач:

*Перевод с одного языка на другой*

Сети передается текстовая строка на исходном языке, а она должна вернуть текст, переведенный на целевой язык.

*Генерирование вопросов*

Сети передается фрагмент текста, а она должна сгенерировать вопрос, который можно было бы задать применительно к тексту.

*Обобщение текста*

Сети передается большой фрагмент текста, а она должна создать краткое изложение.

Для такого рода задач можно использовать сети, известные как *кодировщик-декодировщик*. Мы уже видели пример такой сети, когда исследовали задачу генерирования изображений: вариационный автокодировщик. Для последовательных данных процесс кодирования-декодирования работает следующим образом:

1. Кодировщик RNN преобразует входную последовательность в единый вектор.

2. Этот вектор используется для инициализации декодировщика RNN.

3. Скрытое состояние декодировщика RNN, получаемое на каждом временном шаге, передается в полносвязанный слой, выводящий распределение вероятностей по словарю слов. Так декодировщик, инициализированный представлением исходных данных, созданным кодировщиком, может сгенерировать новую текстовую последовательность.

Для примера на рис. 6.14 показан процесс перевода с английского языка на немецкий.

Окончательное скрытое состояние кодировщика можно рассматривать как представление всего входного документа. Затем декодировщик преобразует это представление в последовательную форму, например, в перевод текста на другой язык или в вопрос, касающийся документа.

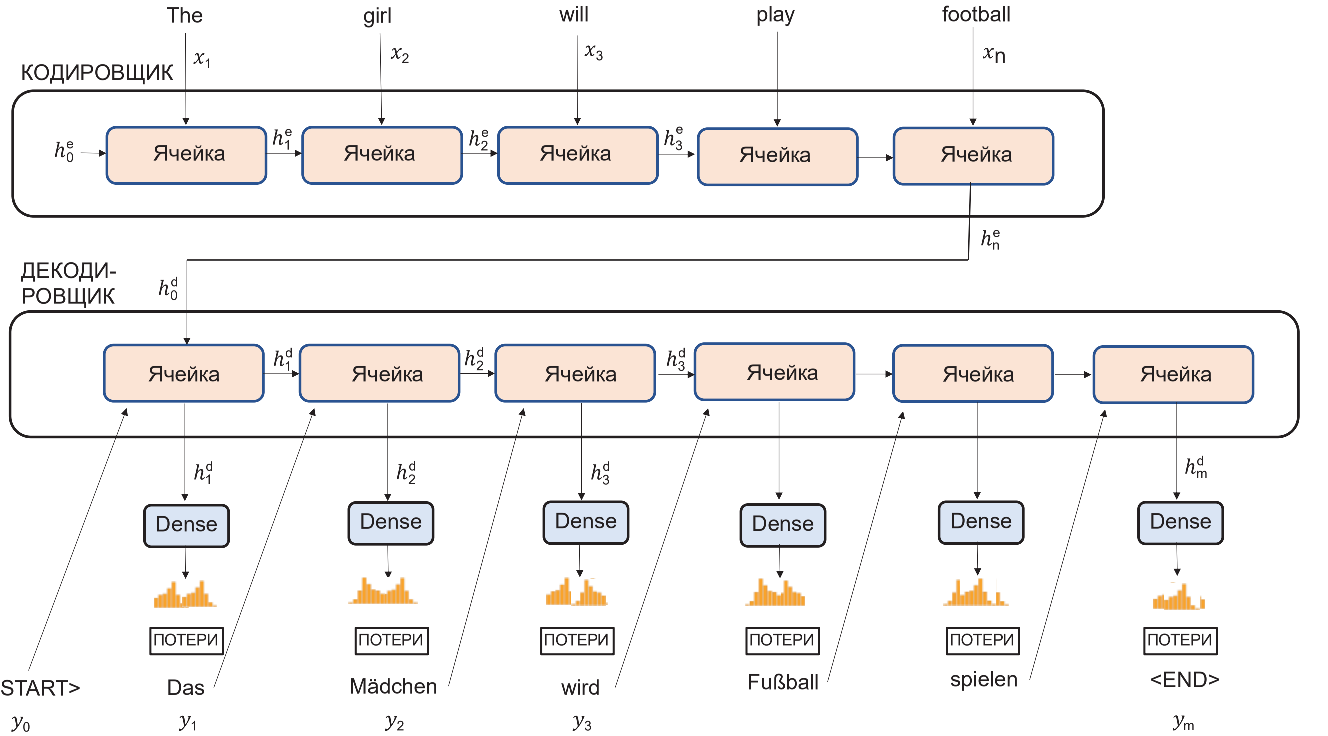



**Рис. 6.14.** Сеть типа кодировщик-декодировщик

Во время обучения выходное распределение, создаваемое декодировщиком на каждом временном шаге, сравнивается со следующим истинным словом для вычисления величины потерь. При обучении декодировщику не нужно делать выбор из этих распределений для получения слов, так как в последующую ячейку подается истинное следующее слово, а не слово, выбранное из предыдущего выходного распределения. Этот способ обучения сетей типа кодировщик-декодировщик известен как *помощь наставника* (teacher forcing). Представьте, что сеть — это ученик, иногда дающий ошибочные прогнозы распределения, но что бы ни вывела сеть на каждом временном шаге, наставник передаст правильный ответ на вход в сеть в виде следующего слова.

## Генератор вопросов и ответов

Теперь давайте соберем все вместе и построим модель, способную генерировать пары вопрос/ответ для блока текста. Прототипом для этого проекта послужили: генератор вопросов и ответов `qgenworkshop` (http://bit.ly/2EUkIg8), использующий TensorFlow, и модель, предложенная Тонг Вангом, Синди Юанем и Адамом Тришлером.[1]

Модель состоит из двух частей:

- сеть RNN, определяющая возможные ответы по содержимому блока текста;
- сеть кодировщика-декодировщика, генерирующая подходящие вопросы для ответов, выделенных сетью RNN.

Например, рассмотрим следующий отрывок из текста с описанием футбольного матча:

```
The winning goal was scored by 23-year-old striker Joe Bloggs during the match
between Arsenal and Barcelona .
Arsenal recently signed the striker for 50 million pounds . The next match is in
two weeks time, on July 31st 2005 . "
```

Перевод:

```
(Победный гол в матче между Арсеналом и Барселоной забил 23-летний нападающий
Джо Блоггс .
```

[1] Тонг Ванг (Tong Wang), Синди Юнь (Xingdi Yuan) и Адам Тришлер (Adam Trischler), «A Joint Model for Question Answering and Question Generation», 5 июля 2017, https://arxiv.org/abs/1706.01450.

```
Недавно Арсенал подписал с нападающим контракт на сумму 50 миллионов фунтов .
Следующий матч состоится через две недели, 31 июля 2005 года . ")
```

Хотелось бы, чтобы наша первая сеть смогла определить потенциальные ответы вроде:

```
"Joe Bloggs"
"Arsenal"
"Barcelona"
"50 million pounds"
"July 31st 2005"
```

Перевод:

```
("Джо Блоггс"
"Арсенал"
"Барселона"
"50 миллионов фунтов"
"31 июля 2005 года")
```

А вторая смогла бы для каждого ответа сгенерировать вопрос, например:

```
"Who scored the winning goal?"
"Who won the match?"
"Who were Arsenal playing?"
"How much did the striker cost?"
"When is the next match?"
```

Перевод:

```
("Кто забил победный гол?"
"Кто победил в матче?"
"С кем Арсенал подписал контракт?"
"Сколько стоит нападающий?"
"Когда состоится следующий матч?")
```

Давайте для начала более детально рассмотрим набор данных, который мы будем использовать.

## Набор данных с вопросами и ответами

Мы будем использовать набор данных Maluuba NewsQA, который можно получить, следуя инструкциям в GitHub (http://bit.ly/2Ky7uJq).

Файлы train.csv, test.csv и dev.csv следует поместить в папку ./data/qa/ внутри репозитория с примерами для книги. Все эти файлы имеют одинаковую структуру, как показано ниже:

`story_id`

Уникальный идентификатор фрагмента текста.

`story_text`

Фрагмент текста (например, «Победный гол в матче между Арсеналом и Барселоной забил 23-летний нападающий Джо Блоггс...»).

`question`

Вопрос, который можно задать к тексту (например, «Сколько стоит нападающий?»).

`answer_token_ranges`

Позиция лексемы в тексте с ответом (например, `24:27`). Может быть указано несколько диапазонов (через запятую), если ответ появляется в тексте несколько раз.

Исходные данные обрабатываются и лексемизируются с тем, чтобы их можно было передать на вход нашей модели. После этого преобразования каждое наблюдение в обучающем наборе включает следующие пять признаков:

`document_tokens`

Лексемизированный текст (например, `[1, 4633, 7, 66, 11, ...]`), усеченный или дополненный нулями до длины `max_document_length` (параметр).

`question_input_tokens`

Лексемизированный вопрос (например, `[2, 39, 1, 52, ...]`), дополненный нулями до длины `max_question_length` (другой параметр).

`question_output_tokens`

Лексемизированный вопрос, смещенный на один временной шаг (например, `[39, 1, 52, 1866, ...]`), дополненный нулями до длины `max_question_length`.

`answer_masks`

Матрица с бинарной маской, имеющая форму `[max_answer_length, max_document_length]`. Значение `[i, j]` матрицы равно 1, если *i*-е слово в ответе на вопрос находится в *j*-м слове документа, и 0 в противном случае.

`answer_labels`

Бинарный вектор с длиной `max_document_length` (например, `[0, 1, 1, 0, ...]`). *i*-й элемент вектора равен 1, если *i*-е слово в документе может считаться частью ответа, и 0 в противном случае.

Рассмотрим архитектуру модели, способной генерировать пары вопрос/ответ для заданного блока текста.

## Архитектура модели

На рис. 6.15 показана общая архитектура создаваемой модели. Не волнуйтесь, если все сказанное выше выглядит для вас пугающим! Она состоит только из элементов, которые мы уже видели, и в этом разделе мы шаг за шагом пройдемся по архитектуре.

Для начала рассмотрим код, создающий часть модели, изображенную в верхней части диаграммы, которая предсказывает, является ли каждое слово в документе частью ответа или нет. Этот код показан в листинге 6.6. Желающие могут следовать за обсуждением, используя блокнот 06_02_qa_train.ipynb из репозитория с примерами для книги.

**Листинг 6.6.** Архитектура модели, генерирующей пары вопрос/ответ

```
from keras.layers import Input, Embedding, GRU, Bidirectional, Dense, Lambda
from keras.models import Model, load_model
import keras.backend as K
from qgen.embedding import glove

#### ПАРАМЕТРЫ ####

VOCAB_SIZE = glove.shape[0] # 9984
EMBEDDING_DIMENS = glove.shape[1] # 100

GRU_UNITS = 100
DOC_SIZE = None
```

```
ANSWER_SIZE = None
Q_SIZE = None

document_tokens = Input(shape=(DOC_SIZE,), name=»document_tokens») ❶

embedding = Embedding(input_dim = VOCAB_SIZE, output_dim = EMBEDDING_DIMENS
    , weights=[glove], mask_zero = True, name = 'embedding') ❷
document_emb = embedding(document_tokens)

answer_outputs = Bidirectional(GRU(GRU_UNITS, return_sequences=True)
    , name = 'answer_outputs')(document__emb) ❸
answer_tags = Dense(2, activation = 'softmax'
    , name = 'answer_tags')(answer_outputs) ❹
```

❶ Лексемы из документа передаются на вход модели. Чтобы описать объем входных данных, мы используем переменную `DOC_SIZE`, но фактически ей присвоено значение `None`. Причина в том, что архитектура модели не зависит от длины входной последовательности: количество ячеек в слое будет адаптироваться к длине входной последовательности, поэтому не требуется указывать ее явно.

❷ Слой `Embedding` инициализируется векторами слов *GloVe* (см. врезку ниже).

❸ Рекуррентный слой `Bidirectional` – двунаправленный управляемый рекуррентный блок (Gated Recurrent Unit, GRU), возвращающий скрытое состояние на каждом временном шаге.

❹ Выходной полносвязанный слой `Dense` получает скрытое состояние на каждом временном шаге и имеет только два узла с функцией активации *softmax*, представляющих вероятность того, что каждое слово является (1) или не является частью ответа (0).

Чтобы использовать набор GloVe в этом примере, загрузите файл glove.6B.100d.txt (векторные представления 6 миллиардов слов, каждое из которых имеет длину 100) с веб-сайта проекта GloVe, а затем запустите следующий сценарий на Python из репозитория книги, чтобы оставить в наборе только слова, присутствующие в обучающем корпусе:

```
python ./utils/write.py
```

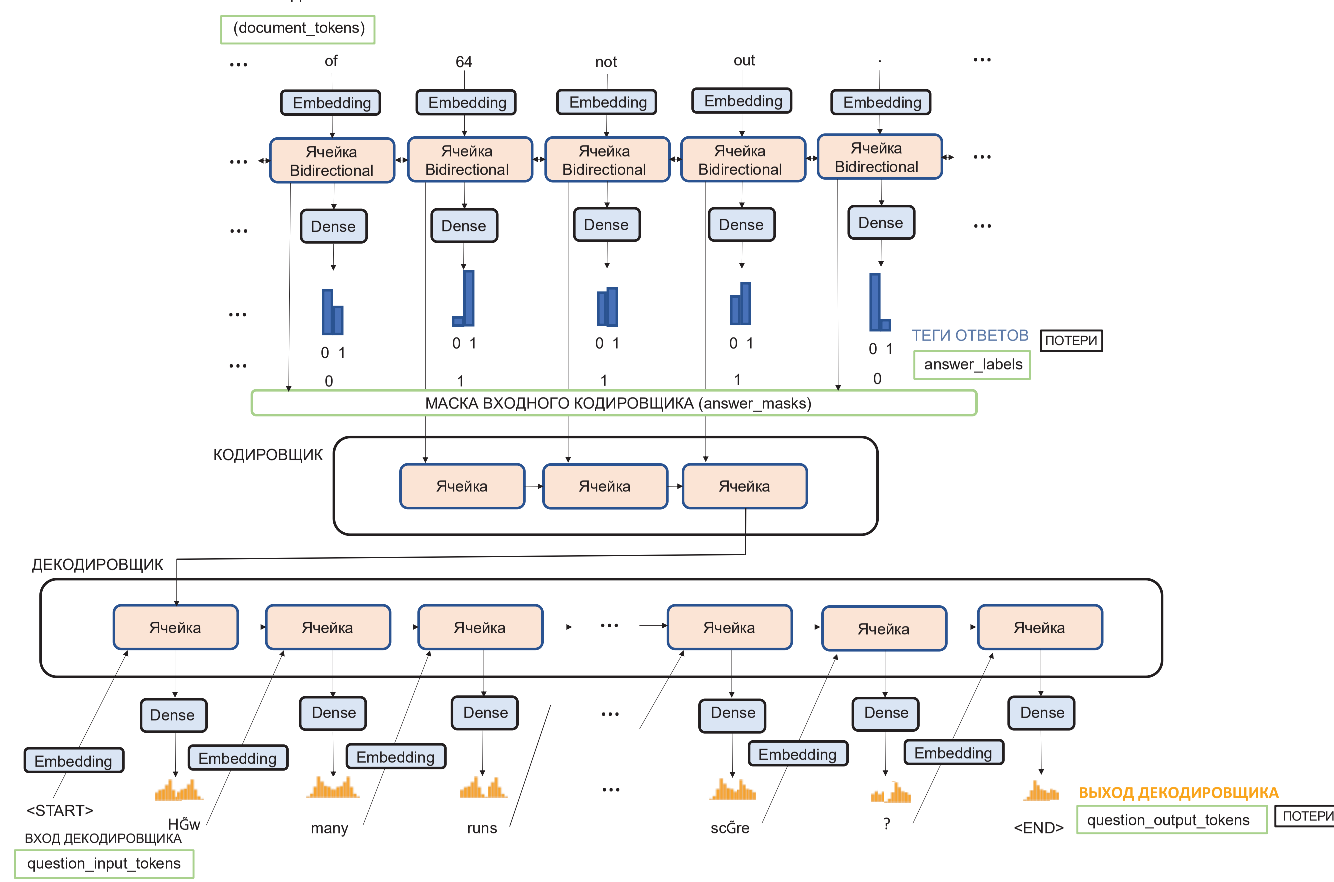



**Рис. 6.15.** Архитектура модели, генерирующей пары вопрос/ответ; входные данные показаны в прямоугольниках с зелеными рамками

**ВЕКТОРЫ СЛОВ GLOVE**

Слой векторного представления `Embedding` инициализируется набором предварительно подготовленных векторных представлений слов, а не случайными векторами, как мы делали это ранее. Эти векторные представления были созданы в рамках проекта Stanford GloVe («Global Vectors») в ходе обучения без учителя с целью получения репрезентативных векторов для большого набора слов.

Эти векторы обладают множеством полезных свойств, таких как сходство векторов родственных слов и др. Например, векторные представления слов *man* (мужчина) и *woman* (женщина) схожи между собой, так же как векторные представления слов *king* (король) и *queen* (королева), как если бы род слов был закодирован в скрытом пространстве, в котором находятся векторы. Инициализация слоя `Embedding` набором GloVe часто позволяет получить лучшие результаты, чем обучение с нуля, поскольку в процессе создания набора GloVe уже проделана большая, тяжелая работа по выявлению высокоуровневых признаков. В результате наш алгоритм может адаптировать векторные представления под свой конкретный контекст.

Вторая часть модели — это сеть кодировщика-декодировщика, которая принимает данный ответ и пытается сформулировать соответствующий вопрос (нижняя часть рис. 6.15).

Код, конструирующий эту часть сети с помощью Keras, показан в листинге 6.7.

**Листинг 6.7.** Архитектура сети кодировщика-декодировщика в модели, которая формулирует вопрос по заданному ответу

```
encoder_input_mask = Input(shape=(ANSWER_SIZE, DOC_SIZE)
    , name=»encoder_input_mask») ❶
encoder_inputs = Lambda(lambda x: K.batch_dot(x[0], x[1])
    , name="encoder_inputs")([encoder_input_mask, answer_outputs])
encoder_cell = GRU(2 * GRU_UNITS, name = 'encoder_cell')(encoder_inputs) ❷

decoder_inputs = Input(shape=(Q_SIZE,), name="decoder_inputs") ❸
decoder_emb = embedding(decoder_inputs) ❹
decoder_emb.trainable = False
decoder_cell = GRU(2 * GRU_UNITS, return_sequences = True, name =
                                                     'decoder_cell')
```

```
decoder_states = decoder_cell(decoder_emb, initial_state = [encoder_cell]) ❺

decoder_projection = Dense(VOCAB_SIZE, name = 'decoder_projection'
        , activation = 'softmax', use_bias = False)
decoder_outputs = decoder_projection(decoder_states) ❻

total_model = Model([document_tokens, decoder_inputs, encoder_input_mask]
        , [answer_tags, decoder_outputs])
answer_model = Model(document_tokens, [answer_tags])
decoder_initial_state_model = Model([document_tokens, encoder_input_mask]
        , [encoder_cell])
```

❶ Маска ответа передается на вход модели — это позволит с помощью слоя `Lambda` передать в кодировщик-декодировщик скрытые состояния из одного диапазона ответов.

❷ Кодировщик — это слой `GRU`, передающий на вход скрытые состояния для диапазона ответов.

❸ Входными данными для декодировщика служит вопрос, соответствующий заданному диапазону ответов.

❹ Лексемы слов в вопросе передаются через тот же слой `Embedding`, который использовался в модели идентификации ответа.

❺ Декодировщик реализован как слой `GRU`, инициализируемый последним скрытым состоянием кодировщика.

❻ Скрытые состояния декодировщика передаются в полносвязанный слой `Dense` для формирования распределения по всему словарю для следующего слова в последовательности.

На этом мы завершаем конструирование сети для генерации пар вопрос/ответ. Чтобы обучить сеть, нужно передать ей на вход текст документа, текст вопроса и маски ответов в пакетном режиме и минимизировать перекрестную энтропию при прогнозировании позиции ответа и слов вопроса, используя одинаковые веса.

## Вычисление результатов

Чтобы протестировать модель на входной последовательности документов, которые она прежде никогда не видела, нужно выполнить следующее:

1. Передать строку с документом в генератор ответов, чтобы получить выборку позиций ответов в документе.

2. Выбрать один из полученных блоков ответов и передать его в генератор вопросов (создать соответствующую маску ответа).

3. Передать документ и маску ответа в кодировщик, чтобы сгенерировать начальное состояние для декодировщика.

4. Инициализировать декодировщик этим начальным состоянием и ввести лексему `<START>`, чтобы сгенерировать первое слово вопроса. Продолжать этот процесс, подавая на вход сгенерированные слова, одно за другим, пока модель не выведет лексему `<END>`.

Как отмечалось выше, во время обучения модель использует помощь наставника в виде ввода истинных слов (вместо следующих предсказанных слов) обратно в ячейку декодировщика. Однако во время вычислений модель должна сама сгенерировать вопрос, поэтому нам нужна возможность передавать назад в ячейку декодировщика предсказанные слова, сохраняя при этом скрытое состояние. Для этого можно определить дополнительную модель Keras (`question_model`), принимающую лексему текущего слова и текущее скрытое состояние декодировщика и выводящую прогнозируемое распределение следующего слова и обновленное скрытое состояние декодировщика (листинг 6.8).

**Листинг 6.8.** Прогнозирующая модель

```
decoder_inputs_dynamic = Input(shape=(1,), name="decoder_inputs_dynamic")
decoder_emb_dynamic = embedding(decoder_inputs_dynamic)
decoder_init_state_dynamic = Input(shape=(2 * GRU_UNITS,)
    , name = 'decoder_init_state_dynamic')
decoder_states_dynamic = decoder_cell(decoder_emb_dynamic
    , initial_state = [decoder_init_state_dynamic])
decoder_outputs_dynamic = decoder_projection(decoder_states_dynamic)

question_model = Model([decoder_inputs_dynamic, decoder_init_state_dynamic]
    , [decoder_outputs_dynamic, decoder_states_dynamic])
```

Эту модель можно использовать в цикле для пословной генерации вопросов (листинг 6.9).

**Листинг 6.9.** Генерация пар вопрос/ответ для данного документа

```
test_data_gen = test_data()
batch = next(test_data_gen)
answer_preds = answer_model.predict(batch[«document_tokens»])

idx = 0
start_answer = 37
end_answer = 39

answers = [[0] * len(answer_preds[idx])]
for i in range(start_answer, end_answer + 1):
    answers[idx][i] = 1

answer_batch = expand_answers(batch, answers)

next_decoder_init_state = decoder_initial_state_model.predict(
    [answer_batch['document_tokens'][[idx]],
    answer_batch['answer_masks'][[idx]]])

word_tokens = [START_TOKEN]
questions = [look_up_token(START_TOKEN)]

ended = False

while not ended:

    word_preds, next_decoder_init_state = question_model.predict(
    [word_tokens, next_decoder_init_state])

    next_decoder_init_state = np.squeeze(next_decoder_init_state, axis = 1)
    word_tokens = np.argmax(word_preds, 2)[0]

    questions.append(look_up_token(word_tokens[0]))

    if word_tokens[0] == END_TOKEN:
        ended = True

questions = ' '.join(questions)
```

## Результаты моделирования

На рис. 6.16 показаны примеры результатов, полученных моделью (см. также блокнот 06_03_qa_analysis.ipynb в репозитории с примерами для книги). Диаграмма справа показывает вероятность каждого слова в документе стать частью ответа. Далее эти фразы с ответами поступают в генератор вопросов, результаты вычислений которого отображаются в левой части диаграммы («Предсказанный вопрос»).

Прежде всего обратите внимание, насколько точно генератор ответов определяет, какие слова из документа вероятнее всего будут содержаться в ответе. Одно это уже впечатляет, особенно если учесть, что он никогда не видел этот текст раньше, а также, возможно, не видел некоторые слова, включенные в ответ, такие как *Bloggs* (Блоггс). Из контекста можно понять, что это, вероятно, фамилия человека, которая, следовательно, может быть частью ответа.

Кодировщик извлекает контекст из каждого из этих ответов, чтобы декодировщик смог сгенерировать подходящие вопросы. Обратите внимание: кодировщик смог понять, что в отношении персоны, упомянутой в первом ответе, 23-year-old striker Joe Bloggs (23-летнего нападающего Джо Блоггса), можно сформулировать вопрос, касающийся его способности забивать голы, и передал этот контекст декодировщику, а тот сгенерировал вопрос «who scored the <UNK> ?» («кто забил <UNK> ?»), а не, например, «кто является президентом?».

Декодер закончил этот вопрос тегом `<UNK>` (неизвестно), но не потому, что не знает, что делать дальше, — он просто предполагает, что слово, которое следует дальше, вероятно, отсутствует в основном словаре. Не стоит удивляться, что модель прибегает к помощи тега `<UNK>` в этом контексте, так как многие из узкоспециализированных слов в оригинальном корпусе могут лексемизироваться таким образом.

Отметьте, что во всех случаях декодировщик выбрал правильный «тип» вопроса — кто, сколько или когда — в зависимости от типа ответа. Тем не менее в результатах наблюдаются некоторые проблемы: например, он сгенерировал вопрос «how much money did he lose?» (сколько денег он потерял?) вместо «how much money was paid for the striker?» (сколько стоит нападающий?). Это вполне объяснимо: декодировщик «видит» только конечное состояние кодировщика и не может обратиться к исходному документу за дополнительной информацией.

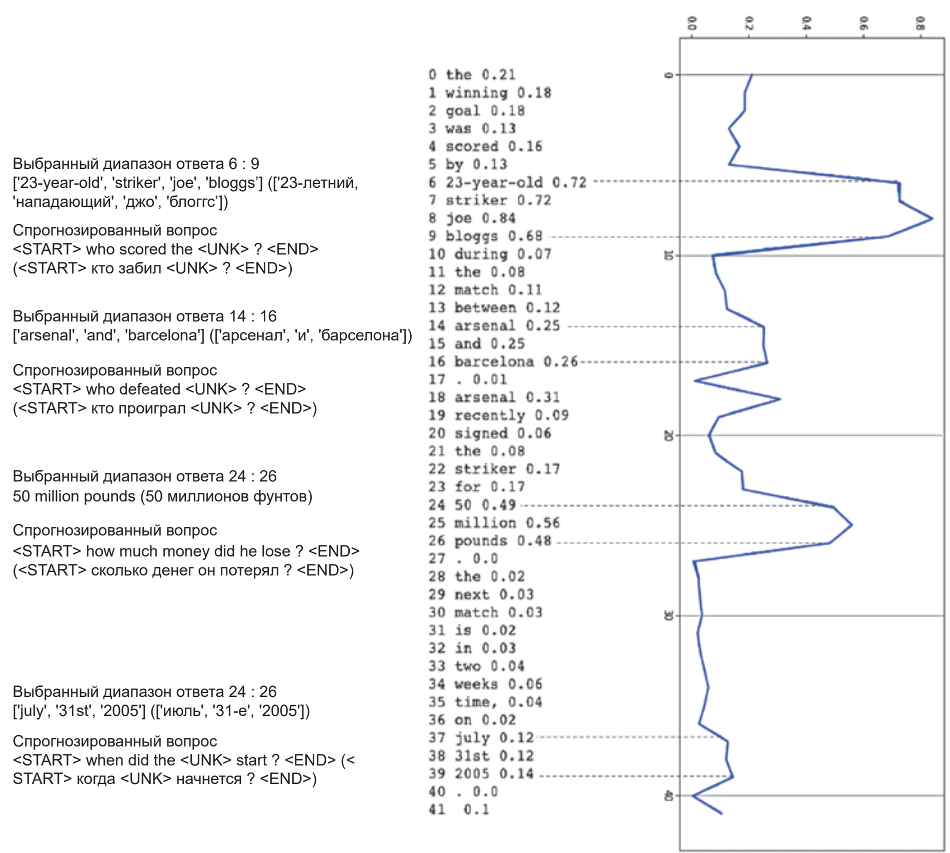



**Рис. 6.16.** Результаты, полученные моделью

Существует несколько расширений сетей кодировщик-декодировщик, которые повышают точность и генеративную мощность модели. Из них наиболее широко используются *сети указателей*[1] и *механизмы внимания*[2]. Сети указателей дают модели возможность «указывать» на конкретные слова во входном тексте для включения в сгенерированный вопрос, а не только полагаться на известные слова в словаре. Это помогает решить проблему

[1] Ориол Виньялс (Oriol Vinyals), Мейре Фортунато (Meire Fortunato) и Навдип Джайтли (Navdeep Jaitly), «Pointer Networks», 9 июля 2015, https://arxiv.org/abs/1506.03134.

[2] Дмитрий Богданау (Dzmitry Bahdanau), Кюнгхён Чо (Kyunghyun Cho) и Йошуа Бенжио (Yoshua Bengio), «Neural Machine Translation by Jointly Learning to Align and Translate», 1 сентября 2014, https://arxiv.org/abs/1409.0473.

с тегом `<UNK>`, упомянутую выше. Мы рассмотрим механизмы внимания в следующей главе.

## Итоги

В этой главе мы выяснили, как применять рекуррентные нейронные сети для генерации текстовых последовательностей, имитирующих определенный стиль письма, а также для создания вероятных пар вопрос/ответ для данного документа. Мы исследовали два разных типа рекуррентных слоев, долгой краткосрочной памяти и GRU, и то, как эти ячейки можно компоновать или делать двунаправленными для формирования сложных сетевых архитектур. Архитектура кодировщик-декодировщик, представленная в этой главе, является важным генеративным инструментом. Она позволяет упаковать последовательные данные в вектор, который затем можно декодировать в другую последовательность. Этот прием применим ко многим задачам, кроме генерации пар вопрос/ответ, таким как перевод с одного языка на другой и обобщение текста.

В обоих случаях мы видели, как важно уметь преобразовывать неструктурированные текстовые данные в структурированный формат, который можно использовать с рекуррентными слоями нейронной сети. Хорошее понимание того, как меняется форма тензора при передаче данных через сеть, тоже имеет большое значение для создания успешных сетей, а рекуррентные слои требуют особой осторожности в этом отношении, потому что временное измерение в последовательных данных добавляет дополнительную сложность в процесс преобразования. В следующей главе речь пойдет о том, как те же идеи, лежащие в основе RNN, можно применить к другому типу последовательных данных — музыке.

ГЛАВА 7

# Сочинение музыки

Сочинение музыки, наряду с живописью и созданием литературных произведений, — одно из ярчайших проявлений творческих способностей человека.

Чтобы сочинять гармоничную, приятную для человеческого слуха музыку, компьютер должен уметь преодолевать технические проблемы сродни рассмотренным в предыдущей главе, выявлять и воссоздавать последовательную музыкальную структуру, а также выбирать последующие ноты из дискретного набора.

Более того, модель, генерирующая музыку, сталкивается с дополнительными проблемами, которые отсутствуют при генерировании текста, а именно определение высоты тональности и ритма создаваемой мелодии. Кроме того, музыка часто полифонична, то есть одновременно звучит несколько потоков нот, извлекаемых на разных инструментах, которые объединяются и создают диссонантные (конфликтующие) или консонантные (согласующиеся) гармонии, поэтому при генерировании музыки, в отличие от текста, требуется обрабатывать не единственный поток слов, а параллельные потоки аккордов.

Текст можно обрабатывать по словам, однако подобный метод сочинения плохо подходит для работы с музыкальными данными, потому что наибольший эмоциональный настрой вызывает прослушивание музыки, получающейся объединением различных ритмов в ансамбле. Например, гитарист может играть быстро сменяющиеся ноты, тогда как пианист способен воспроизводить более длинные выдержанные аккорды. Проще говоря, создание музыки нота за нотой — сложное дело, потому что часто бывает нежелательно, чтобы все инструменты меняли ноту одновременно.

Мы начнем эту главу с того, что упростим задачу и сосредоточимся на генерировании музыки для одной (монофонической) музыкальной линии. Как вы увидите сами, многие методы RNN, описанные в предыдущей главе, где рассказывалось о генерировании текста, тоже могут использоваться для генерирования музыки, поскольку эти две задачи имеют много общего. В этой главе также будет представлен механизм внимания (attention mechanism), позволяющий создавать рекуррентные сети, способные выбирать, на каких предыдущих нотах сосредоточиться, чтобы предсказать, какие ноты должны следовать далее. Наконец, мы рассмотрим задачу создания полифонической музыки и узнаем, как развернуть архитектуру, основанную на GAN, с целью создания музыки для нескольких голосов.

## Вступление

Приступая к решению задачи генерирования мелодий, необходимо прежде получить общее представление о теории музыки. Рассмотрим основные формы записи музыки и то, как представить музыку в «цифре», чтобы передать ее в таком виде на вход генеративной модели для обучения.

Здесь нас вновь выручит блокнот 07_01_notation_compose.ipynb из репозитория с примерами для книги. Другим ценным ресурсом для желающих заняться генерированием музыки на Python может послужить статья (http://bit.ly/2XtmRXr) в блоге Сигурдура Скули и сопутствующий репозиторий в GitHub (http://bit.ly/2I07hgv).

В качестве исходных данных в этой главе мы используем набор MIDI-файлов с сочинениями И. С. Баха для виолончели (в блокноте вы найдете все необходимые инструкции по загрузке). Желающие могут использовать иной набор данных по своему выбору.

Для прослушивания музыки, сгенерированной моделью, понадобится программное обеспечение, способное создавать нотную запись, например MuseScore, доступный бесплатно.

### Нотная запись

Для загрузки и обработки MIDI-файлов используем библиотеку для Python `music21`. Листинг 7.1 демонстрирует, как загрузить MIDI-файл и изобразить его (рис. 7.1) в виде партитуры и структурированных данных.

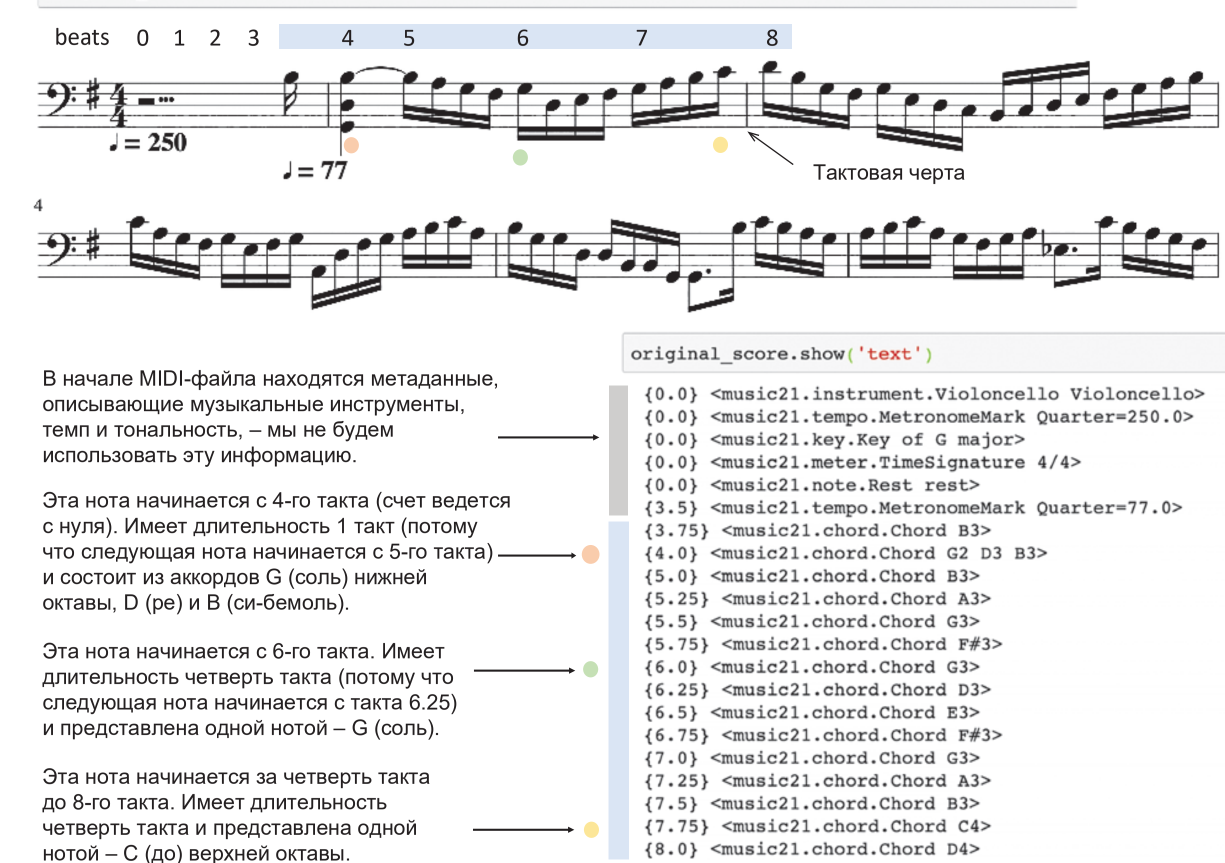



**Рис. 7.1.** Нотная запись

**Листинг 7.1.** Импортирование MIDI-файла

```
from music21 import converter

dataset_name = 'cello'
filename = 'cs1-2all'
file = "./data/{}/{}.mid".format(dataset_name, filename)

original_score = converter.parse(file).chordify()
```

Чтобы сжать все одновременно проигрываемые ноты в аккорды воедино, используется метод `chordify`. Поскольку эта пьеса исполняется одним инструментом (виолончелью), мы имеем полное право на это, хотя иногда бывает желательно оставить отдельные части, чтобы сгенерировать полифоническую музыку. Это создает дополнительные сложности, которые мы преодолеем далее в этой главе.

Код в листинге 7.2 перебирает в цикле содержимое партитуры и извлекает высоту и длительность каждой ноты (и пауз) в два списка. Отдельные ноты в аккордах разделены точкой, поэтому весь аккорд можно сохранить как одну строку. Число после каждого названия ноты указывает ее *октаву* — поскольку имена нот (от A до G)[1] повторяются в каждой октаве, это дополнительное число помогает однозначно определить высоту ноты. Например, `G2` на октаву ниже, чем `G3`.

**Листинг 7.2.** Извлечение данных

```
notes = []
durations = []

for element in original_score.flat:

    if isinstance(element, chord.Chord):
        notes.append('.'.join(n.nameWithOctave for n in element.pitches))
        durations.append(element.duration.quarterLength)

if isinstance(element, note.Note):
    if element.isRest:
        notes.append(str(element.name))
        durations.append(element.duration.quarterLength)

    else:
        notes.append(str(element.nameWithOctave))
        durations.append(element.duration.quarterLength)
```

[1] В английском языке используются названия нот: A, B, H, C, D, E, F, G — ля, си-бемоль, си, до, ре, ми, фа, соль. — *Примеч. пер.*

Полученный набор данных (табл. 7.1) теперь больше похож на текст, с которым мы имели дело ранее. Слова представляют *звуки*, и мы должны попытаться построить модель, предсказывающую следующий звук, исходя из предыдущей последовательности звуков. Эту же идею можно применить к списку длительностей. Keras дает нам возможность сконструировать модель, способную прогнозировать высоту тона и длительность одновременно.

**Таблица 7.1.** Высота и длительность каждой ноты, хранящейся в списке

| Длительность | Высота | Длительность | Высота |
|---|---|---|---|
| 0,25 | B3 | 0,25 | G3 |
| 1,0 | G2.D3.B3 | 0,25 | D3 |
| 0,25 | B3 | 0,25 | E3 |
| 0,25 | A3 | 0,25 | F#3 |
| 0,25 | G3 | 0,25 | G3 |
| 0,25 | F#3 | 0,25 | A3 |

## Ваша первая сеть RNN для генерирования музыки

Чтобы создать набор данных для обучения модели, сначала нужно присвоить каждой ноте и длительности целочисленное значение (рис. 7.2), как мы делали это для слов в текстовом корпусе. Неважно, какими будут эти значения, так как мы будем использовать слой векторного представления для преобразования целочисленных значений в векторы.

```
note_to_int                    duration_to_int

{'A2': 0,                      {0: 0,
 'A2.A3': 1,                    Fraction(1, 12): 1,
 'A2.B2': 2,                    Fraction(1, 6): 2,
 'A2.C3': 3,                    0.25: 3,
 'A2.D3': 4,                    Fraction(1, 3): 4,
 'A2.E-3': 5,                   0.5: 5,
 'A2.E3': 6,                    Fraction(2, 3): 6,
 'A2.E3.A3': 7,                 0.75: 7,
 'A2.E3.C#4': 8,                1.0: 8,
 'A2.E3.C#4.A4': 9,             1.25: 9,
 'A2.E3.C#4.E4': 10,            Fraction(4, 3): 10,
```

**Рис. 7.2.** Словари для преобразования нот и длительностей в целочисленные значения

Затем мы создадим обучающий набор, разбив данные на небольшие блоки по 32 ноты с переменной ответа, содержащей следующую ноту в последовательности (в формате прямого кодирования), то есть высоту тона и длительность звучания (рис. 7.3).

Модель, которую мы построим, представляет собой многослойную сеть LSTM с механизмом внимания. В предыдущей главе мы видели, как объединять слои LSTM и передавать скрытое состояние предыдущего слоя на вход следующего. Такой способ объединения слоев позволяет модели выявлять более сложные признаки в данных. В этом разделе мы познакомимся с механизмом внимания[1], который в настоящее время является неотъемлемой частью сложных последовательных генеративных моделей. Появление этого механизма породило *трансформеры* — тип моделей, основанных исключительно на внимании и не требующих наличия рекуррентных или сверточных

```
pitch input
[386  77 340  77  27 371 260 371 168 237 260 371  27  77 144 196  77 371
 260 371 237 168 124  61 124 168 237 260 371  27  77 144]
duration input
[0 3 8 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3]
pitch output
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0.]
duration output
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

**Рис. 7.3.** Выходные и выходные данные генеративной музыкальной модели

[1] Дмитрий Богданау (Dzmitry Bahdanau), Кюнгнхён Чо (Kyunghyun Cho) и Йошуа Бенжио (Yoshua Bengio), «Neural Machine Translation by Jointly Learning to Align and Translate», 1 сентября 2014, https://arxiv.org/abs/1409.0473.

слоев. Более подробно архитектура трансформеров будет обсуждаться в главе 9, а пока сосредоточимся на внедрении механизма внимания в многослойную сеть LSTM, попробовав предсказать следующую ноту, основываясь на последовательности предыдущих нот.

## Внимание

Первоначально механизм внимания применялся в задачах перевода текста с одного языка на другой, в частности с английского на французский.

В предыдущей главе мы видели, как сети типа «кодировщик-декодировщик» решают эту задачу, пропуская сначала входную последовательность через кодировщик, чтобы сгенерировать вектор контекста, который затем возвращается через сеть декодировщика в виде перевода. Одна из проблем такого подхода — в том, что контекстный вектор может стать узким местом. Информация из исходного предложения может оказаться существенно разбавленной, достигнув вектора контекста, что особенно верно для длинных предложений. Поэтому иногда разработчики применяют дополнительные механизмы, помогающие сохранить всю необходимую информацию для более точного перевода исходного текста декодировщиком.

Например, модель должна перевести следующее предложение на немецкий: *I scored a penalty in the football match against England* (я забил пенальти в футбольном матче против Англии).

Очевидно, что смысл этого предложения в корне изменится, если заменить слово *scored* (забил) на *missed* (пропустил), причем есть вероятность, что кодировщик не сможет сохранить эту информацию в конечном скрытом состоянии, так как слово *scored* находится в начале предложения.

Правильный перевод предложения: *Ich habe im Fußballspiel gegen England einen Elfmeter erzielt.*

Как видим, в переводе на немецкий язык слово *scored* (*erzielt*) действительно находится в конце предложения! То есть информация о том, что пенальти был забит, а не пропущен, должна сохраниться при прохождении не только через кодировщик, но и через декодировщик.

То же верно и в отношении музыки. Для определения ноты или последовательности нот, которая, скорее всего, последует за конкретным отрывком, вполне может пригодиться информация из далекого начала отрывка, а не

только из близлежащего фрагмента. Например, вот вступительный отрывок из прелюдии к сюите № 1 Баха для виолончели (рис. 7.4).

Какая нота должна следовать дальше? Даже если у вас нет музыкального образования, вы все равно сможете догадаться. Сказав «соль» (первая нота в этом произведении), вы будете совершенно правы — возможно, заметив, что каждый такт и половина такта начинаются с одной и той же ноты, и использовав этот факт для обоснования своего решения. Хотелось бы, чтобы наша модель смогла проделать тот же трюк, то есть чтобы она заботилась не только о *текущем* скрытом состоянии сети, но и уделяла особое внимание скрытому состоянию, имевшему место восемь нот тому назад, когда была встречена предыдущая нота G (соль) нижней октавы.

Для решения этой проблемы был предложен механизм внимания. Вместо единственного конечного скрытого состояния для создания вектора контекста механизм внимания использует взвешенную сумму скрытых состояний кодировщика RNN, имевших место на всех предыдущих временных шагах. Механизм внимания — это просто набор слоев, которые преобразуют предыдущие скрытые состояния кодировщика и текущее скрытое состояние декодировщика во взвешенные суммы, на основе которых генерируется вектор контекста.

Если это объяснение показалось вам слишком сложным, не волнуйтесь! Мы начнем с того, что добавим механизм внимания после простого рекуррентного слоя (чтобы решить задачу прогнозирования следующей ноты в сюите № 1 Баха для виолончели), и только потом будем рассматривать, как этот прием применить к сети кодировщик-декодировщик, чтобы предсказать уже не одну, а целую последовательность нот.

**Рис. 7.4.** Вступительный отрывок из прелюдии к сюите № 1 Баха для виолончели

## Конструирование механизма внимания с помощью Keras

Для начала вспомним, как использовать стандартный рекуррентный слой для прогнозирования следующей ноты, опираясь на последовательность предыдущих нот. На рис. 7.5 показано, как входная последовательность ($x_1$, ..., $x_n$) подается нота за нотой в слой, который на каждом шаге обновляет свое скрытое состояние. Входная последовательность может состоять из векторных представлений нот или скрытых состояний из предыдущего рекуррентного слоя. На выходе рекуррентного слоя получается конечное скрытое состояние, вектор которого имеет длину, равную количеству узлов. Далее этот вектор можно передать в слой `Dense` с выходом *softmax*, чтобы предсказать наиболее вероятную следующую ноту.

На рис. 7.6 показана та же сеть, но уже с механизмом внимания для учета предыдущих скрытых состояний рекуррентного слоя.

Давайте рассмотрим, шаг за шагом, как действует эта сеть (листинг 7.3):

1. Сначала каждое скрытое состояние $h_j$ (вектор, длина которого равна количеству узлов в рекуррентном слое) передается в функцию $a$, которую называют функцией *согласования*, чтобы получить скаляр $e_j$. В этом примере функция согласования реализована как обычный полносвязанный слой с одним выходным узлом и функцией активации *tanh*.

2. Далее, к вектору $e_1$, ..., $e_n$ применяется функция *softmax*, чтобы получить вектор весов $\alpha_1$, ..., $\alpha_n$.

3. Наконец, каждый вектор скрытого состояния $h_j$ умножается на соответствующий вес $\alpha_j$ и результаты суммируются, чтобы получить вектор контекста $c$ (соответственно, $c$ имеет ту же длину, что и вектор скрытого состояния).

Полученный вектор контекста можно передать в слой *Dense* с выходом *softmax*, как обычно, чтобы получить распределение вероятностей для следующей потенциальной ноты.

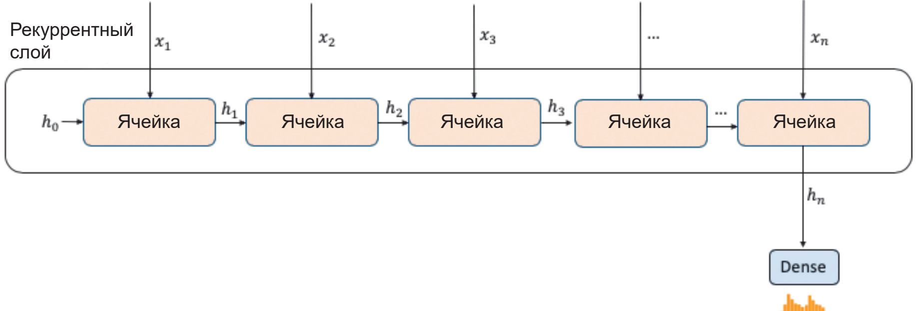
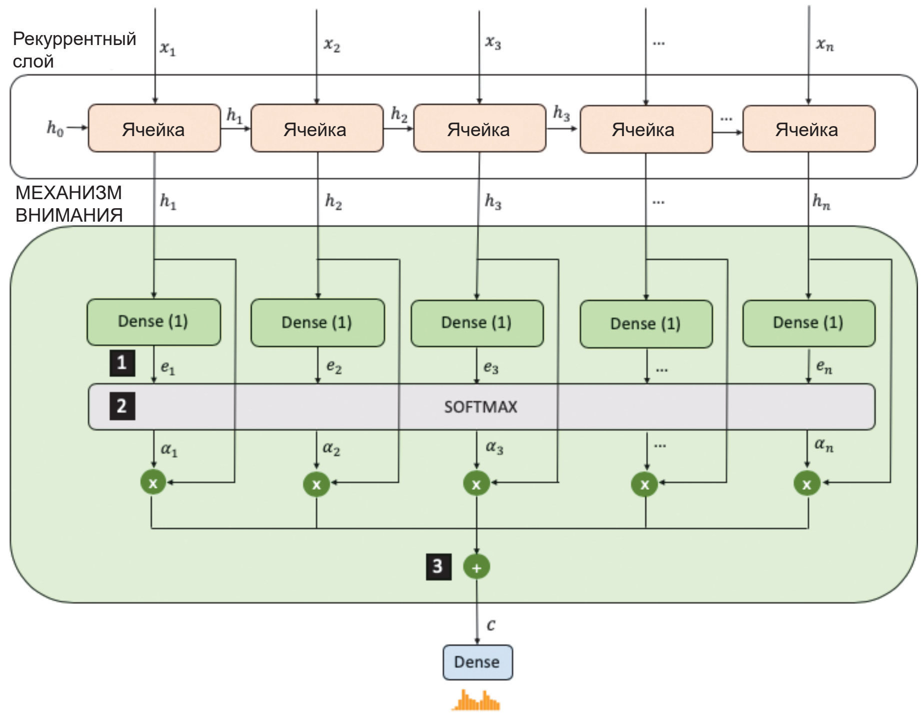



**Рис. 7.5.** Рекуррентный слой для предсказания следующей ноты в последовательности без механизма внимания

**Листинг 7.3.** Сборка RNN с механизмом внимания

```
notes_in = Input(shape = (None,)) ❶
durations_in = Input(shape = (None,))

x1 = Embedding(n_notes, embed_size)(notes_in) ❷
x2 = Embedding(n_durations, embed_size)(durations_in)

x = Concatenate()([x1,x2]) ❸
x = LSTM(rnn_units, return_sequences=True)(x) ❹
x = LSTM(rnn_units, return_sequences=True)(x)

e = Dense(1, activation='tanh')(x) ❺
e = Reshape([-1])(e)

alpha = Activation('softmax')(e) ❻

c = Permute([2, 1])(RepeatVector(rnn_units)(alpha)) ❼
c = Multiply()([x, c])
c = Lambda(lambda xin: K.sum(xin, axis=1), output_shape=(rnn_units,))©

notes_out = Dense(n_notes, activation = 'softmax', name = 'pitch')© ❽
durations_out = Dense(n_durations, activation = 'softmax', name = 'duration')©

model = Model([notes_in, durations_in], [notes_out, durations_out]) ❾

att_model = Model([notes_in, durations_in], alpha) ❿

opti = RMSprop(lr = 0.001)
model.compile(loss=['categorical_crossentropy', 'categorical_crossentropy'],
                optimizer=opti) ⓫
```




**1** $e_j = a(\boldsymbol{h}_j) = \tanh(W \cdot \boldsymbol{h}_j)$

**2** $\alpha_j = [softmax(\boldsymbol{e})]_j = \dfrac{\exp(e_j)}{\sum_k \exp(e_k)}$

**3** $\boldsymbol{c} = \sum_j \alpha_j \boldsymbol{h}_j$

**Рис. 7.6.** Слой для предсказания следующей ноты в последовательности с механизмом внимания

❶ Сеть имеет два входа: последовательность названий предыдущих нот и значения их длительности. Обратите внимание: длина последовательности не указана — механизм внимания не требует, чтобы входная последовательность имела фиксированную длину, поэтому мы можем пренебречь ею.

❷ Слои `Embedding` преобразуют целочисленные значения, соответствующие названиям нот и длительности нот, в векторы.

❸ Из двух векторов формируется один большой вектор для передачи в рекуррентные слои.

❹ Рекуррентную часть сети образуют два слоя LSTM. Обратите внимание: здесь мы передаем параметр `return_sequence` со значением `True`, чтобы оба слоя выводили полные последовательности скрытых состояний, а не только конечное скрытое состояние.

❺ Функция согласования — это обычный слой `Dense` с одним выходным узлом и функцией активации *tanh*. Слой `Reshape` используется для сжатия выходных данных в единственный вектор, длина которого равна длине входной последовательности (`seq_length`).

❻ Веса вычисляются применением функции активации *softmax* к согласованным значениям.

❼ Чтобы получить взвешенную сумму скрытых состояний, используем слой `RepeatVector`, который копирует веса в `rnn_units` и создает матрицу с формой `[rnn_units, seq_length]`. Затем транспонируем эту матрицу с помощью слоя `Permute`, чтобы получить матрицу с формой `[seq_length, rnn_units]`. Потом поэлементно умножим матрицу на скрытые состояния из конечного слоя LSTM, который также имеет форму `[seq_length, rnn_units]`. Наконец, используем слой `Lambda` для суммирования по оси `seq_length`, чтобы получить контекстный вектор с длиной `rnn_units`.

❽ Сеть имеет двусторонний выход: название следующей ноты и ее длительность.

❾ Окончательная модель принимает названия предыдущих нот и их длительности и выводит распределение вероятностей для выбора названия следующей ноты и ее продолжительности.

❿ Дополнительно создается модель, которая выводит вектор слоя `alpha`, чтобы можно было понять, как сеть определяет веса для предыдущих скрытых состояний.

⓫ Модель компилируется с использованием функций потерь `categoryorical_crossentropy` для названий и длительности нот, потому что эта задача относится к категории многоклассовой классификации.

На рис. 7.7 показана диаграмма полной модели, сконструированной с помощью Keras.Обучить эту сеть LSTM с механизмом внимания можно, запустив блокнот 07_02_lstm_compose_train.ipynb из репозитория с примерами для книги.

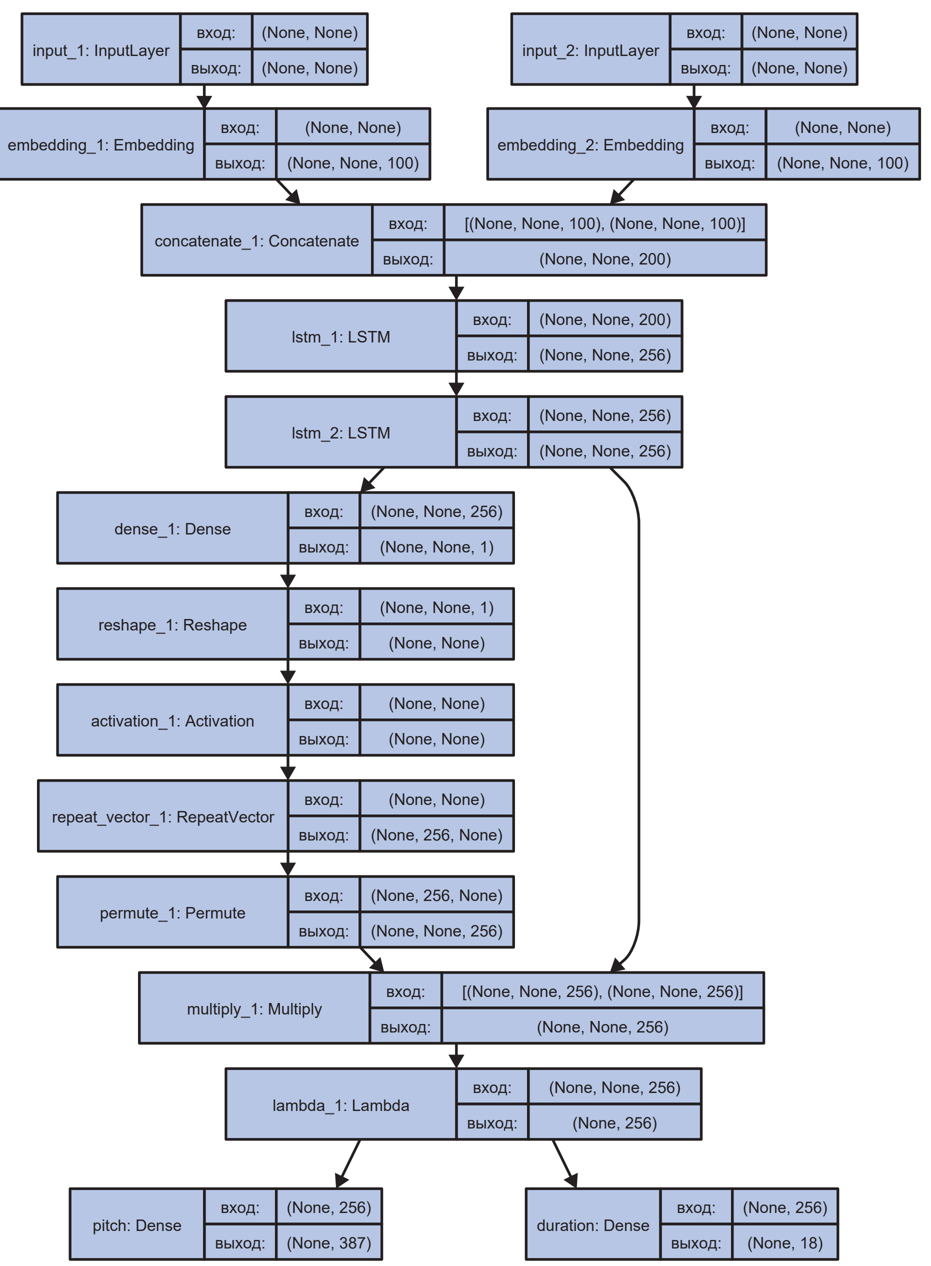



**Рис. 7.7.** Модель LSTM с механизмом внимания для предсказания следующей ноты в последовательности

## Анализ сети RNN с механизмом внимания

Анализ, описываемый далее, можно выполнить, запустив блокнот 07_03_lstm_compose_analysis.ipynb из репозитория с примерами для книги после обучения сети.

Чтобы сгенерировать музыкальный фрагмент с нуля, передадим сети последовательность с единственным тегом `<START>` (то есть сообщим модели, что она должна начать с самого начала). Затем сгенерируем музыкальный фрагмент, используя тот же способ, что использовался в главе 6 для генерирования текстовых последовательностей:

1. На основе текущей последовательности (с названиями нот и их длительностями) модель спрогнозирует два распределения вероятностей для выбора следующей ноты и ее продолжительности.
2. Выберем значения из обоих распределений, используя параметр `temperature`, который управляет долей случайности при выборе.
3. Сохраним выбранную ноту, а ее название и продолжительность добавим в соответствующие последовательности.
4. Если длина сгенерированной последовательности превысила длину обучающей последовательности, то удалим один элемент из начала последовательности.
5. Процесс повторяется с каждой новой последовательностью до получения фрагмента желаемой длины.

Примеры мелодий, сгенерированных моделью с нуля, при разных количествах эпох обучения, представлены на рис. 7.8.

Основное внимание в этом разделе мы уделим предсказаниям тональности нот, потому что сюиты Баха для виолончели имеют очень сложный гармонический строй, ритмику которого трудно выявить, что требует гораздо более тщательного обучения сети. Тем не менее те же приемы анализа, что описываются далее, можно применить и к ритмическим прогнозам модели, что может быть особенно актуально для других музыкальных стилей, которые могут быть использованы для обучения модели (например, ударных инструментов).

Следует также отметить несколько моментов, касающихся сгенерированных фрагментов на рис. 7.8. Во-первых, с каждой эпохой обучения музыка все больше усложняется. В самом начале модель стремится воспроизводить все те же группы нот и ритмов. К эпохе 10 модель начала генерировать небольшие серии нот, а к эпохе 20 – производить интересные ритмы, прочно закрепившись в выборе тональности (ми-бемоль мажор).

Во-вторых, мы можем проанализировать распределение высот нот во времени, построив тепловую карту прогнозируемого распределения на каждом временном шаге. Пример тепловой карты для эпохи 20, изображенной на рис. 7.8, представлен на рис. 7.9.

**Рис. 7.8.** Некоторые примеры музыкальных фрагментов, сгенерированных моделью, на вход которой был подан единственный тег <START>; здесь использовался параметр temperature со значением 0,5, управляющий выбором названий нот и их продолжительности

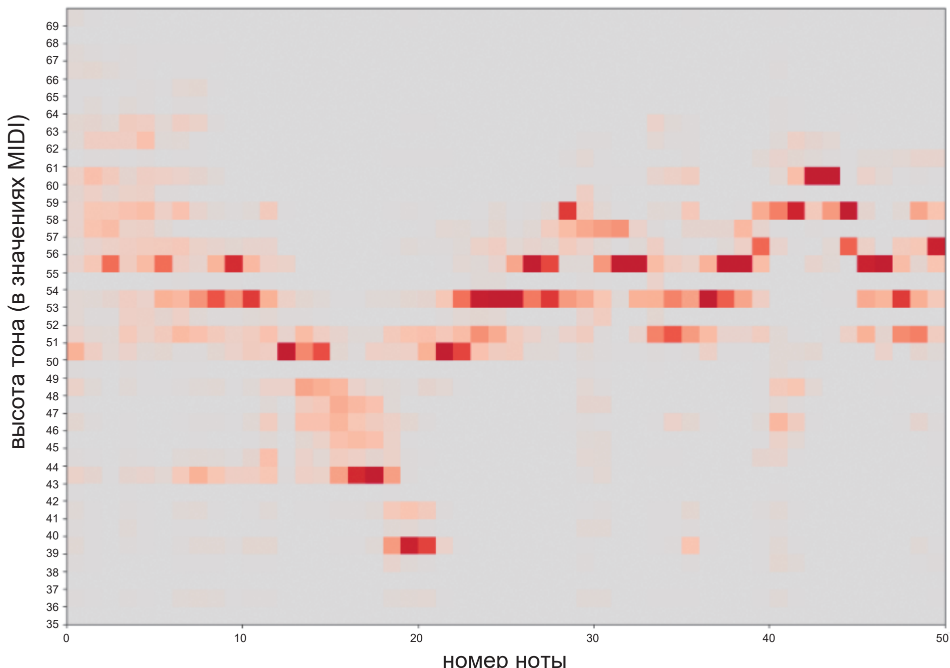



**Рис. 7.9.** Распределение вероятностей возможных следующих нот в эпоху 20: чем темнее квадратик, тем выше вероятность, что модель выберет эту ноту в качестве следующей

Любопытно, что модель четко определила, какие ноты принадлежат конкретным *тональностям*. Об этом свидетельствуют пробелы в распределениях, соответствующих нотам, не принадлежащим тональности. Например, в ряду для ноты 54 (соль-бемоль/фа-мажор) имеется серый разрыв. Эта нота вряд ли появится в музыкальном произведении, имеющем тональность ми-бемоль мажор. На ранней стадии процесса (левая часть диаграммы) тональность еще не определена, и поэтому существует большая неопределенность в выборе следующей ноты. По мере продвижения модель выбирает определенную тональность, и поэтому некоторые ноты почти не имеют шанса быть выбранными. Особенно примечательно, что изначально модель не выбирала тональность для генерируемой музыки, а буквально придумала ее, пытаясь подобрать ноту, которая лучше всего соответствует тем, что были выбраны ранее.

Стоит также отметить, что модель выучила характерный для Баха стиль понижения тона в конце фразы с последующим резким переходом на высокую ноту. Посмотрите на окружение ноты 20. Эта фраза заканчивается низким ми-бемоль — в своих сюитах для виолончели в начале следующей фразы Бах обычно возвращается к более высокому, более звучному диапазону инструмента, что очень точно предсказывает модель. Между низким ми-бемоль (высота тона 39) и следующей предсказанной нотой (с высотой тона 50) имеется большой серый промежуток, что говорит об отсутствии переборов в низкой тональности.

Наконец, давайте проверим, как работает наш механизм внимания. На рис. 7.10 показаны значения элементов альфа-вектора, вычисленные сетью в каждой точке сгенерированной последовательности. Горизонтальная ось соответствует сгенерированной последовательности нот; вертикальная ось показывает, куда было направлено внимание сети при прогнозировании каждой ноты на горизонтальной оси (то есть альфа-вектор). Чем темнее квадратик, тем больше внимания уделяется скрытому состоянию, соответствующему этой точке последовательности.

Как видим, на второй ноте во фрагменте (`B-3` = си-бемоль) сеть решила почти все свое внимание уделить тому факту, что первой нотой пьесы тоже была нота `B-3`. В этом есть определенный смысл: зная, что первая нота — си-бемоль, вы почти наверняка воспользуетесь этой информацией при выборе следующей ноты. Далее сеть примерно одинаково распределяет свое внимание между предыдущими нотами, весьма редко выделяя ноты, отстоящие более чем на шесть нот назад. И в этом есть определенный смысл: скорее всего, в предыдущих шести скрытых состояниях содержится достаточно информации для того, чтобы понять, как эта музыкальная фраза должна продолжаться.

Есть также примеры, когда сеть решила игнорировать определенную близкорасположенную ноту, потому что та не добавляет дополнительной информации к пониманию фразы. Например, посмотрите на прямоугольник с белыми границами в центре диаграммы и обратите внимание, что в середине есть полоса прямоугольников, нарушающая сложившийся шаблон заглядывания назад на 4–6 нот. Почему сеть охотно предпочла бы игнорировать эту ноту, принимая решение о выборе продолжения фразы? Посмотрев, какой ноте эта полоса соответствует, вы увидите, что это — первая из трех нот `E-3` (ми-бемоль). Модель предпочла проигнорировать ее, потому что ей предшествует та же нота ми-бемоль, но на октаву ниже (`E-2`). Скрытое состояние сети

на этом этапе предоставляет модели достаточно информации, чтобы понять, что ми-бемоль является важной нотой в этом отрывке, и поэтому модели нет необходимости обращать внимание на последующую более высокую ноту ми-бемоль, так как та не добавляет никакой дополнительной информации.

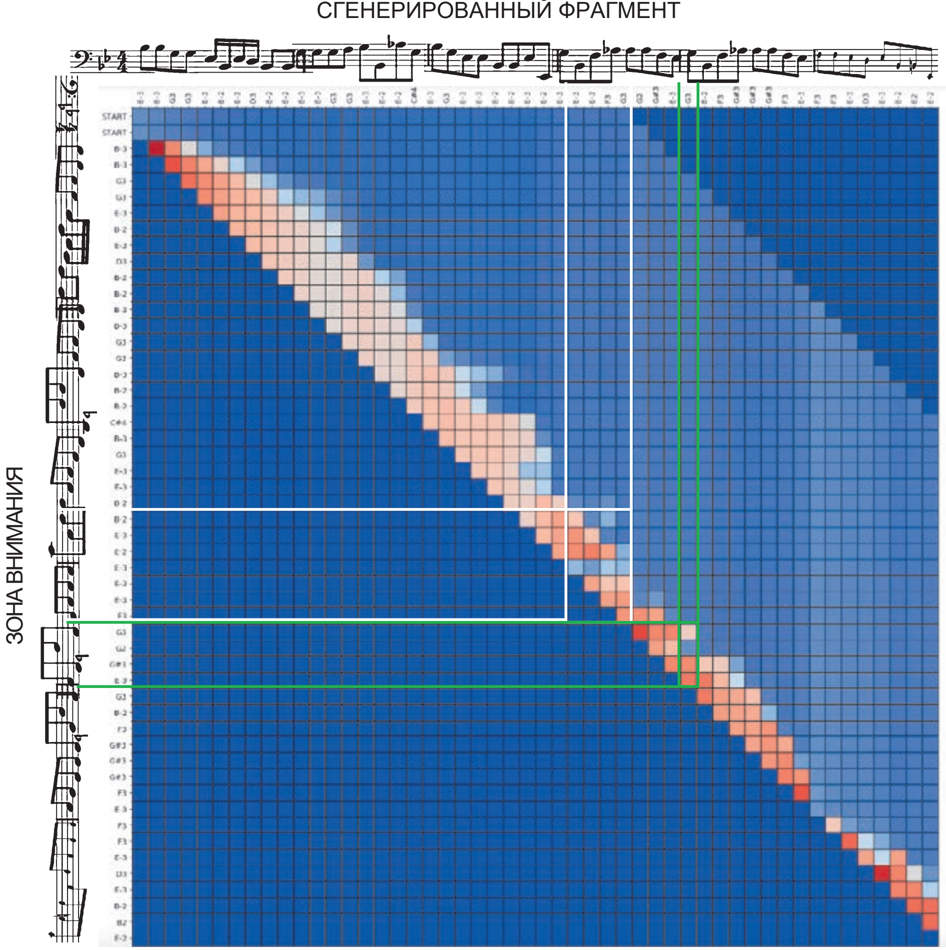



**Рис. 7.10.** Цвет каждого квадратика отражает количество внимания, уделявшегося скрытому состоянию сети, которое соответствует ноте на вертикальной оси в точке прогнозирования на горизонтальной оси; чем более насыщенный красный цвет имеет квадратик, тем больше внимания уделялось

Дополнительные доказательства того, что модель начала понимать идею октавы, можно увидеть внутри прямоугольника с зеленой рамкой, ниже и правее. Здесь модель решила игнорировать соль нижней октавы (`G2`), потому что ей предшествовала та же нота соль (`G3`), но на октаву выше. Как вы помните, мы ничего не сообщали модели о том, как связаны ноты через октавы, — модель сама обнаружила это, просто изучив музыку И. С. Баха, что особенно примечательно.

## Механизм внимания в сетях типа кодировщик-декодировщик

Механизм внимания — мощный инструмент, помогающий сети решить, какие предыдущие состояния рекуррентного уровня важны для прогнозирования продолжения последовательности. До сих пор мы использовали его только для прогнозирования на одну ноту вперед. Но механизм внимания можно внедрить и в сеть типа кодировщик-декодировщик, прогнозирующую последовательность будущих нот с помощью декодировщика RNN, вместо того чтобы выбирать ноты по одной. На рис. 7.11 показано, как может выглядеть стандартная модель кодировщик-декодировщик для генерирования музыки без механизма внимания (с сетью этого вида мы познакомились в главе 6, а на рис. 7.12 показана та же сеть, но с механизмом внимания между кодировщиком и декодировщиком).

Механизм внимания в этом случае работает точно так же, как описано выше, с одним исключением: скрытое состояние декодировщика также передается в механизм, поэтому модель может сосредоточить внимание не только на предыдущих скрытых состояниях кодировщика, но и на текущем скрытом состоянии декодировщика. На рис. 7.13 показано внутреннее устройство модуля механизма внимания в структуре кодировщик-декодировщик.

Несмотря на то что в сети типа кодировщик-декодировщик имеется много копий механизма внимания, все они используют одни и те же весовые коэффициенты, поэтому накладные расходы на обучение не зависят от числа изучаемых параметров. Единственное отличие в том, что теперь в вычислениях механизма внимания участвует и скрытое состояние декодировщика (красные линии на диаграмме). Это немного меняет уравнения, которые теперь включают дополнительный индекс ($i$), определяющий шаг декодировщика.

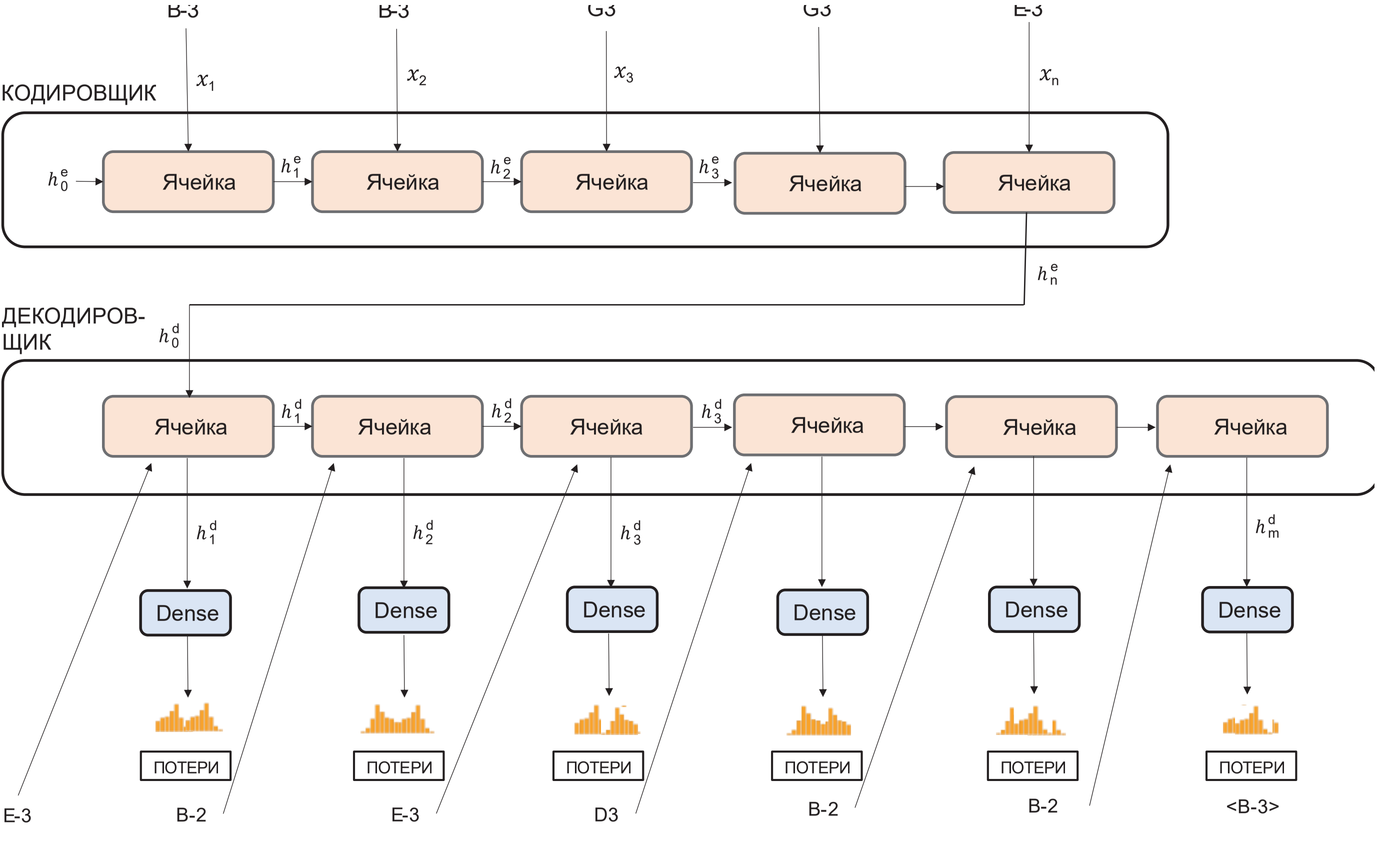



**Рис. 7.11.** Стандартная модель типа кодировщик-декодировщик

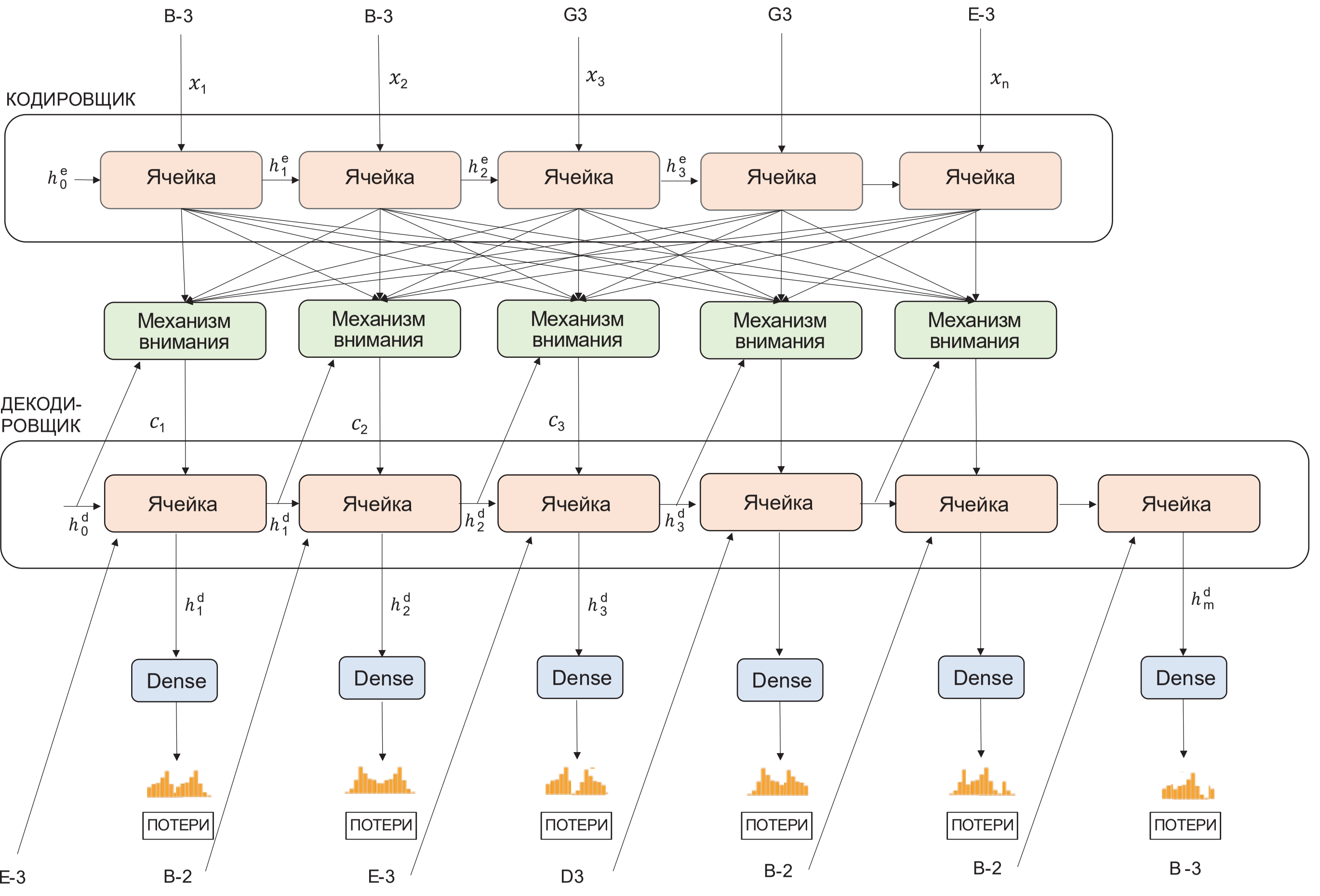



**Рис. 7.12.** Модель типа кодировщик-декодировщик с механизмом внимания

На рис. 7.11 скрытое состояние декодировщика инициализируется конечным состоянием кодировщика. В сетях типа кодировщик-декодировщик с механизмом внимания декодировщик инициализируется с помощью встроенных стандартных инициализаторов рекуррентного слоя. Для каждой ячейки декодировщика создается расширенный вектор данных, получаемый путем объединения вектора контекста $c_i$ со входными данными $y_{i-1}$. Таким образом, векторы контекста можно рассматривать как дополнительные данные, передаваемые в декодировщик.

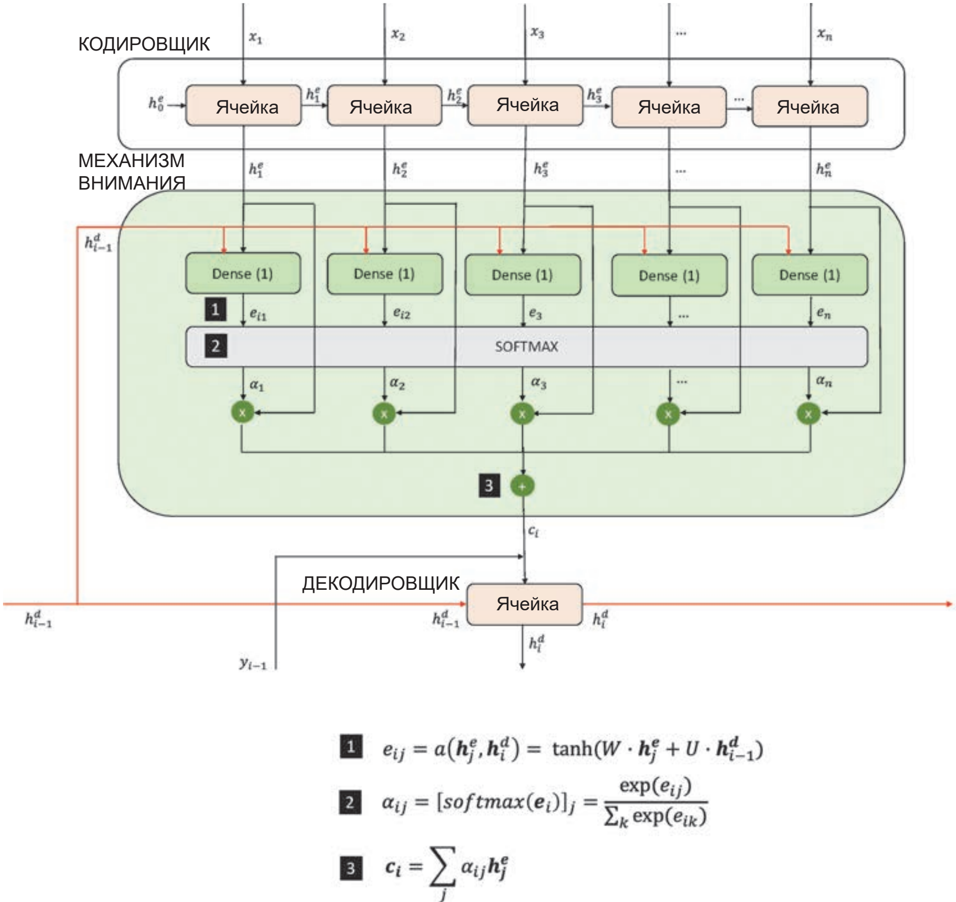



**Рис. 7.13.** Механизм внимания в контексте сети типа кодировщик-декодировщик, подключенный к декодировщику ячейки i

### Генерирование полифонической музыки

Инфраструктура RNN с механизмом внимания, которую мы исследовали в этом разделе, прекрасно справляется с задачей генерирования монофонической музыки. А можно ли адаптировать ее для генерирования полифонии? Инфраструктура RNN, безусловно, обладает достаточной гибкостью и позволяет создать архитектуру, генерирующую сразу несколько музыкальных линий. Но наш текущий набор данных плохо подходит для этого, потому что аккорды хранятся в нем как отдельные объекты, а не как части, состоящие из нескольких отдельных нот. Наша текущая сеть RNN не способна понять, например, что аккорд до-мажор (до, ми и соль) на самом деле очень близок к аккорду ля-минор (ля, до и ми) — достаточно заменить лишь одну ноту: соль на ля. Вместо этого она считает их двумя отдельными элементами, которые могут предсказываться независимо. В идеале нам нужна сеть, которая принимает несколько музыкальных каналов в виде отдельных потоков и изучает, как эти потоки взаимодействуют друг с другом, чтобы потом генерировать красивую музыку, а не дисгармоничный шум.

Для генерирования изображений, напомним, использовались три канала цвета (красный, зеленый и синий), и сеть училась комбинировать их для создания образов, а не случайного пиксельного шума. Фактически, как мы увидим в следующем разделе, задачу генерирования музыки можно считать прямым аналогом задачи генерирования изображений. Это означает, что вместо рекуррентных сетей можно использовать те же методы на основе свертки, которые так хорошо зарекомендовали себя в задачах генерирования изображений в музыке, в частности, в сетях GAN. Но прежде чем приступить к исследованию новой архитектуры, посетим концертный зал, где начинается концерт.

## Музыкальный орган

Дирижер дважды постучал палочкой по пюпитру. Концерт начинается. Но на сцене — оркестр, исполняющий не симфонии Бетховена и не увертюры Чайковского. Этот оркестр сочиняет оригинальную музыку и состоит из исполнителей, посылающих команды огромному музыкальному органу (назовем его MuseGAN), который превращает эти команды в прекрасную музыку, доставляющую удовольствие слушателям. Оркестр можно научить генерировать музыку в определенном стиле, причем ни одно из его выступлений не похоже на другое. Сто двадцать восемь ис-

полнителей в оркестре разделены на четыре равные секции по 32 исполнителя. Каждая секция дает команды органу MuseGAN и имеет свои обязанности в оркестре.

Секция *стиля* отвечает за создание общего музыкального стиля исполнения. Во многих отношениях она выполняет самую простую работу: каждый исполнитель этой секции должен сгенерировать только одну команду в начале концерта и затем непрерывно подавать ее в MuseGAN на протяжении всего выступления.

Секция *дорожек* выполняет похожую работу, но каждый исполнитель в ней выдает уже несколько команд: по одной для каждой музыкальной *партии*, которая выводится органом MuseGAN. Например, в одном концерте каждый участник секции дорожек подготовил пять команд, по одной для каждой из пяти партий: вокальной, фортепианной, струнной, басовой и партии ударных инструментов. Их работа заключается в том, чтобы обеспечить *дорожку* для каждого отдельного инструмента, которая будет присутствовать на протяжении всего исполнения.

Секции стиля и дорожек не меняют своих команд на протяжении всего исполнения. Динамику в исполнение привносят остальные две группы, обеспечивающие изменение мелодии в каждом такте. *Такт* (или *цифра*) — это единичный фрагмент мелодии, содержащий небольшое и фиксированное количество *долей*. Например, если, слушая музыку, вы можете сосчитать 1, 2, 1, 2, то это значит, что каждый такт состоит из двух долей и, скорее всего, это марш. Если вы можете сосчитать 1, 2, 3, 1, 2, 3, значит, каждый такт состоит из трех долей и вы слушаете вальс.

Исполнители в секции *аккордов* меняют свои команды в начале каждого такта. Это придает каждому такту особый музыкальный характер. Каждый исполнитель в этой секции посылает одну команду в начале каждого такта, которая затем применяется к каждой инструментальной дорожке.

Исполнители в секции *мелодии* выполняют самую утомительную работу: дают разные команды для каждой инструментальной дорожки в начале каждого такта. Они имеют наиболее полный контроль над музыкой, и поэтому их можно рассматривать как секцию обеспечения мелодической линии.

Закончив знакомство с организацией оркестра, обобщим обязанности каждой секции (табл. 7.2).

**Таблица 7.2.** Секции оркестра MuseGAN

| | Команды изменяются в каждом такте? | Разные команды в разных дорожках? |
|---|---|---|
| Стиль | × | × |
| Дорожки | × | ✓ |
| Аккорды | ✓ | × |
| Мелодия | ✓ | ✓ |

Каждый следующий такт MuseGAN должен сгенерировать с учетом текущего набора из 128 команд (по одной от каждого исполнителя). Обучить MuseGAN этому нелегко. Первоначально инструмент производит только ужасающий шум, так как не может понять, как интерпретировать команды и создавать из них такты, неотличимые от настоящей музыки. На помощь органу приходит дирижер. Дирижер сообщает ему, когда звуки, которые он воспроизводит, явно отличаются от настоящей музыки, после чего MuseGAN адаптирует свои внутренние связи, чтобы в следующий раз с большей вероятностью удовлетворить ожидания дирижера. Дирижер и MuseGAN следуют тому же процессу, который мы видели в главе 4, когда Ди и Джин работали вместе над улучшением фотографий животных, сделанных Джином. Исполнители, управляющие органом MuseGAN, путешествуют по миру, давая концерты в любом стиле, если получают достаточно продолжительный музыкальный фрагмент для обучения MuseGAN. В следующем разделе мы увидим, как создать MuseGAN с помощью Keras и научить его генерировать реалистичную полифоническую музыку.

## Ваша первая сеть MuseGAN

Сеть MuseGAN представлена в статье «MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment».[1] В ней авторы показали, как с помощью новейшего фреймворка GAN обучить модель генерировать полифоническую музыку и, разделяя

[1] Хао-Вен Донг (Hao-Wen Dong) и др., «MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment», 19 сентября 2017, https://arxiv.org/abs/1709.06298.

обязанности между векторами шума, которые питают генератор, обеспечить полный контроль над высокоуровневыми признаками музыки.

Чтобы опробовать этот пример, сначала нужно скачать MIDI-файлы, на которых будет обучаться сеть MuseGAN. Используем набор данных из 229 хоралов И. С. Баха для четырех голосов, доступный на GitHub (http://bit.ly/2HYISrC). Загрузим набор и поместим его в папку data в репозитории с примерами для книги (в подпапку chorales). Набор данных состоит из массива, содержащего по четыре числа для каждого временного шага — высоту ноты MIDI для каждого из четырех голосов. Временной шаг в этом наборе равен одной шестнадцатой ноты. Так, например, в одном такте с 4 четвертями будет 16 временных шагов. Кроме того, набор данных автоматически делится на *обучающий*, *проверочный* и *контрольный*. Для обучения MuseGAN используем *обучающий* набор. Сначала данные нужно привести к форме, пригодной для передачи в GAN. Сгенерируем два такта музыки и поэтому сначала извлечем из каждого хорала первые два такта (см. порядок преобразования исходных данных в векторы для передачи в сеть GAN

и соответствующую им музыкальную нотацию на рис. 7.14). Каждый такт делится на 16 временных шагов, то есть всего для четырех партий — 84 шага. Следовательно, подходящая форма для преобразованных данных имеет вид:

```
[batch_size, n_bars, n_steps_per_bar, n_pitches, n_tracks]
```

где

```
n_bars = 2
n_steps_per_bar = 16
n_pitches = 84
n_tracks = 4
```

Чтобы привести данные к этой форме, закодируем числовые значения нот в вектор с длиной 84 и разобьем каждую последовательность нот на две группы по шестнадцать, чтобы получить два такта.[1] Подготовив данные, давайте рассмотрим общую структуру MuseGAN, начиная с генератора.

[1] Предполагается, что каждый хорал имеет четыре доли в каждом такте, что вполне разумно, и даже если бы это было не так, то не оказало бы отрицательного влияния на обучение модели.

ИСХОДНЫЕ ДАННЫЕ

ПРЕОБРАЗОВАННЫЕ ДАННЫЕ

4 дорожки

2 такта

16 временных шагов

```
array([[74., 70., 65., 58.],
       [74., 70., 65., 58.],
       [74., 70., 65., 58.],
       [74., 70., 65., 58.],
       [75., 70., 58., 55.],
       [75., 70., 58., 55.],
       [75., 70., 60., 55.],
       [75., 70., 60., 55.],
       [77., 69., 62., 50.],
       [77., 69., 62., 50.],
       [77., 69., 62., 50.],
       [77., 69., 62., 50.],
       [77., 70., 62., 55.],
       [77., 70., 62., 55.],
       [77., 69., 62., 55.],
       [77., 69., 62., 55.],
       [75., 67., 63., 48.],
       [75., 67., 63., 48.],
       [75., 69., 63., 48.],
       [75., 69., 63., 48.],
       [74., 70., 65., 46.],
       [74., 70., 65., 46.],
       [74., 70., 65., 46.],
       [74., 70., 65., 46.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
```

4 дорожки

84 ноты

16 временных шагов

2 такта

МУЗЫКАЛЬНАЯ ПАРТИТУРА

2 такта

16 временных шагов

**Рис. 7.14.** Пример исходных данных для обучения MuseGAN

## Генератор MuseGAN

Подобно всем генеративно-состязательным сетям (GAN), MuseGAN состоит из генератора и критика. Генератор пытается одурачить критика своими музыкальными творениями, а критик старается предотвратить это, пытаясь отличить поддельные хоралы Баха, созданные генератором, от настоящих. Особенность сети MuseGAN в том, что ее генератор принимает на входе не один вектор шума, а целых четыре, которые соответствуют четырем секциям оркестра (см. выше) — аккордов, стиля, мелодии и дорожек (голосов). Управляя каждым из этих входов, можно менять высокоуровневые свойства сгенерированной музыки (рис. 7.15).

На диаграмме видно, как входные тензоры аккордов и мелодии сначала передаются в промежуточную сеть, которая выводит тензор, одно из измерений которого равно количеству сгенерированных тактов. Благодаря этому нет необходимости растягивать во времени входные тензоры стилей и дорожек, так как они остаются постоянными на протяжении всего произведения. Затем, чтобы сгенерировать конкретный такт для конкретной партии, объединяются соответствующие векторы аккордов, стиля, мелодии и дорожек. Они формируют более длинный вектор, передаваемый в генератор тактов, который в конечном итоге выводит указанный такт для указанной партии. Объединением сгенерированных тактов для всех партий создается партитура, которую критик может сравнить с настоящими результатами. Запустить обучение MuseGAN можно с помощью блокнота 07_04_musegan_train.ipynb из репозитория с примерами для книги (листинг 7.4).

**Листинг 7.4.** Определение MuseGAN

```
BATCH_SIZE = 64
n_bars = 2
n_steps_per_bar = 16
n_pitches = 84
n_tracks = 4
z_dim = 32

gan = MuseGAN(input_dim = data_binary.shape[1:]
, critic_learning_rate = 0.001
, generator_learning_rate = 0.001
, optimiser = 'adam'
, grad_weight = 10
, z_dim = 32
, batch_size = 64
, n_tracks = 4
, n_bars = 2
, n_steps_per_bar = 16
, n_pitches = 84
)
```

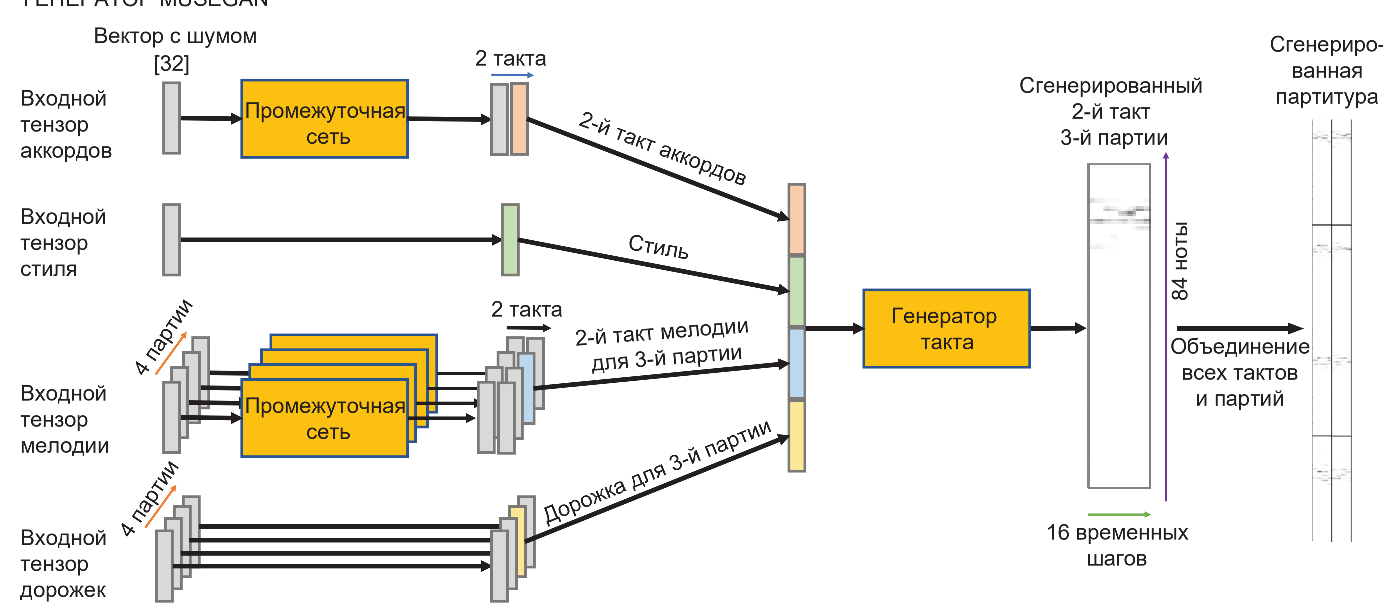



**Рис. 7.15.** Обобщенная схема генератора MuseGAN

## Аккорды, стиль, мелодия и дорожки

Теперь рассмотрим поближе четыре входных вектора, которые передаются в генератор.

### Аккорды

Входной тензор аккордов имеет длину 32 (`z_dim`). Мы должны получить разные векторы для разных тактов, потому что их задача — обеспечить динамическую изменчивость музыки с течением времени. Обратите внимание: несмотря на свое имя `chords_input`, этот тензор может контролировать все, что касается изменения музыки с началом нового такта, например общий ритмический стиль, без привязки к какой-либо конкретной партии. Это достигается с помощью нейронной сети, состоящей из слоев обратной свертки, которую мы называем *промежуточной сетью*. Определение этой сети с помощью Keras показано в листинге 7.5.

**Листинг 7.5.** Конструирование промежуточной сети

```
def conv_t(self, x, f, k, s, a, p, bn):
    x = Conv2DTranspose(
                filters = f
                , kernel_size = k
                , padding = p
                , strides = s
                , kernel_initializer = self.weight_init
                )(x)

    if bn:
      x = BatchNormalization(momentum = 0.9)(x)

    if a == 'relu':
        x = Activation(a)(x)
    elif a == 'lrelu':
        x = LeakyReLU()(x)

    return x

def TemporalNetwork(self):

    input_layer = Input(shape=(self.z_dim,), name='temporal_input') ❶

    x = Reshape([1,1,self.z_dim])(input_layer) ❷
    x = self.conv_t(x, f=1024, k=(2,1), s=(1,1), a= 'relu',
                    p = 'valid', bn = True)
    x = self.conv_t(x, f=self.z_dim, k=(self.n_bars – 1,1)
      , s=(1,1), a= 'relu', p = 'valid', bn = True) ❸
```

```
    output_layer = Reshape([self.n_bars, self.z_dim])(x) ❹

    return Model(input_layer, output_layer)
```

❶ На входе промежуточная сеть принимает вектор с длиной 32 (`z_dim`).

❷ Этот вектор преобразуется в тензор с формой 1 × 1 и 32 каналами, чтобы к нему можно было применить операцию обратной свертки.

❸ Слои `Conv2DTranspose` увеличивают размер тензора вдоль одной из осей, чтобы он получил длину, равную `n_bars`.

❹ С помощью слоя `Reshape` удаляются ставшие ненужными дополнительные измерения.

Используем сверточные операции вместо передачи двух независимых векторов тактов, с тем чтобы сеть знала, что такты должны следовать друг за другом. Использование нейронной сети для расширения входного вектора вдоль оси времени позволяет модели узнать, что музыка исполняется тактами и каждый следующий такт не является полностью независимым от предыдущего.

### Стиль

Входной тензор стиля тоже имеет длину `z_dim`. Он передается в генератор тактов без каких-либо изменений, потому что для него не существует понятий партии и такта. Другими словами, генератор тактов должен использовать этот тензор для согласования тактов и партий.

### Мелодия

Входной тензор мелодии — это массив с формой `[n_tracks, z_dim]`, и для каждой партии модели передается случайный вектор шума с длиной `z_dim`. Каждый из этих векторов передается через свою копию промежуточной сети, описанной выше. Обратите внимание: веса в этих копиях разные, и поэтому на выходе для каждой партии в каждом такте получается вектор с длиной `z_dim`. При подобной организации генератор тактов может использовать этот вектор для точной настройки содержимого каждого отдельного такта в каждой партии.

### Дорожки

Входной тензор дорожек тоже является массивом с формой [n_tracks, z_dim], то есть массивом, содержащим вектор со случайным шумом и длиной z_dim для каждой партии. В отличие от тензора мелодии, он не преобразуется промежуточной сетью, а передается непосредственно в генератор тактов, как тензор стиля. Но, в отличие от тензора стиля, каждой партии соответствует отдельная входная дорожка. Таким образом, эти векторы можно использовать для независимой настройки каждой партии.

## Генератор тактов

Генератор тактов преобразует вектор с длиной 4 * z_dim в один такт для одной партии, то есть в тензор с формой [1, n_steps_per_bar, n_pitches, 1]. Входной вектор создается объединением четырех соответствующих векторов аккордов, стилей, мелодий и дорожек, каждый из которых имеет длину z_dim. Генератор тактов — это нейронная сеть, которая использует слои обратной свертки для увеличения размерностей, соответствующих времени и нотам. Мы создадим для каждой партии свой генератор тактов, и все генераторы будут иметь свои веса. Код в листинге 7.6 показывает, как с помощью Keras построить генератор тактов.

**Листинг 7.6.** Конструирование генератора тактов

```
def BarGenerator(self):
    input_layer = Input(shape=(self.z_dim * 4,), name='bar_generator_input') ❶

    x = Dense(1024)(input_layer)
    x = BatchNormalization(momentum = 0.9)(x)
    x = Activation('relu')(x)

    x = Reshape([2,1,512])(x) ❷
    x = self.conv_t(x, f=512, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True) ❸
    x = self.conv_t(x, f=256, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True)
    x = self.conv_t(x, f=256, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True)
    x = self.conv_t(x, f=256, k=(1,7), s=(1,7), a= 'relu', p = 'same',bn = True) ❹
    x = self.conv_t(x, f=1, k=(1,12), s=(1,12), a= 'tanh', p = 'same', bn = False) ❺

    output_layer = Reshape([1, self.n_steps_per_bar , self.n_pitches ,1])(x) ❻

    return Model(input_layer, output_layer)
```

❶ На вход генератора тактов подается вектор с длиной `4 * z_dim`.

❷ Пропустив тензор через слой `Dense`, мы подготавливаем его к обратной сверточной операции, изменяя форму.

❸ Сначала тензор расширяется по оси временных шагов...

❹ ...а затем по оси нот.

❺ В последнем слое используется функция активации *tanh*, потому что далее для обучения сети используется механизм WGAN-GP (требующий активации *tanh*).

❻ Форма тензора изменяется, чтобы подготовить его к объединению с другими тактами и партиями, добавлением двух дополнительных измерений с размером 1.

## Объединяем все вместе

В целом сеть MuseGAN имеет единственный генератор, включающий в себя все промежуточные сети и генераторы тактов. Эта сеть принимает четыре входных тензора и преобразует их в партитуру с несколькими тактами и партиями (листинг 7.7).

**Листинг 7.7.** Конструирование генератора MuseGAN

```
chords_input = Input(shape=(self.z_dim,), name='chords_input') ❶
style_input = Input(shape=(self.z_dim,), name='style_input')
melody_input = Input(shape=(self.n_tracks, self.z_dim), name='melody_input')
groove_input = Input(shape=(self.n_tracks, self.z_dim), name='groove_input')

# АККОРДЫ -> ПРОМЕЖУТОЧНАЯ СЕТЬ ❷
self.chords_tempNetwork = self.TemporalNetwork()
self.chords_tempNetwork.name = 'temporal_network'
chords_over_time = self.chords_tempNetwork(chords_input) # [n_bars, z_dim]

# МЕЛОДИЯ -> ПРОМЕЖУТОЧНАЯ СЕТЬ ❸
melody_over_time = [None] * self.n_tracks # list of n_tracks [n_bars, z_dim]
                                                              tensors
self.melody_tempNetwork = [None] * self.n_tracks
for track in range(self.n_tracks):
    self.melody_tempNetwork[track] = self.TemporalNetwork()
    melody_track = Lambda(lambda x: x[:,track,:])(melody_input)
    melody_over_time[track] = self.melody_tempNetwork[track](melody_track)

# СОЗДАНИЕ ГЕНЕРАТОРА ТАКТОВ ДЛЯ КАЖДОЙ ПАРТИИ ❹
```

```
self.barGen = [None] * self.n_tracks
for track in range(self.n_tracks):
    self.barGen[track] = self.BarGenerator()

# СОЗДАНИЕ ВЫХОДА ДЛЯ КАЖДОГО ТАКТА И ПАРТИИ ❺
bars_output = [None] * self.n_bars
for bar in range(self.n_bars):
    track_output = [None] * self.n_tracks

    c = Lambda(lambda x: x[:,bar,:]
        , name = 'chords_input_bar_' + str(bar))(chords_over_time)
    s = style_input

    for track in range(self.n_tracks):

        m = Lambda(lambda x: x[:,bar,:])(melody_over_time[track])
        g = Lambda(lambda x: x[:,track,:])(groove_input)

        z_input = Concatenate(axis = 1
        , name = 'total_input_bar_{}_track_{}'.format(bar, track)
        )([c,s,m,g])

        track_output[track] = self.barGen[track](z_input)

    bars_output[bar] = Concatenate(axis = -1)(track_output)

generator_output = Concatenate(axis = 1, name = 'concat_bars')(bars_output)❻

self.generator = Model([chords_input, style_input, melody_input, groove_input]
        , generator_output) ❼
```

❶ Определить входные слои генератора.

❷ Организовать передачу входного тензора аккордов через промежуточную сеть.

❸ Организовать передачу входного тензора мелодии через промежуточную сеть.

❹ Создать независимую сеть генератора тактов для каждой партии.

❺ Выполнить обход партий и тактов и создать генерируемый такт для каждой комбинации.

❻ Объединить все вместе и сформировать общий выходной тензор.

❼ Модель MuseGAN принимает четыре тензора с шумом и выводит сгенерированную партитуру.

## Критик

По сравнению с генератором критик имеет более простую архитектуру (что довольно характерно для сетей GAN). Критик пытается отличить партитуры, созданные генератором, от фрагментов настоящих хоралов Баха. Это сверточная нейронная сеть, состоящая в основном из слоев `Conv3D`, которые свертывают партитуру в один выходной прогноз. До сих пор мы работали только со слоями `Conv2D`, применяя их к изображениям, имеющим три измерения: ширину, высоту и каналы цвета. Здесь мы должны использовать слои `Conv3D`. Они подобны слоям `Conv2D`, но принимают четырехмерные тензоры (`n_bars`, `n_steps_per_bar`, `n_pitches`, `n_tracks`).

Кроме того, мы не используем слои пакетной нормализации в критике, потому что для обучения GAN будем использовать инфраструктуру WGAN-GP, которая запрещает нормализацию.

В листинге 7.8 показан код, конструирующий критика с помощью Keras.

**Листинг 7.8.** Конструирование критика MuseGAN

```
def conv(self, x, f, k, s, a, p):
    x = Conv3D(
                filters = f
                , kernel_size = k
                , padding = p
                , strides = s
                , kernel_initializer = self.weight_init
                )(x)

    if a =='relu':
        x = Activation(a)(x)
    elif a== 'lrelu':
        x = LeakyReLU()(x)

    return x


critic_input = Input(shape=self.input_dim, name='critic_input') ❶

x = critic_input
x = self.conv(x, f=128, k = (2,1,1), s = (1,1,1), a = 'lrelu', p = 'valid') ❷
x = self.conv(x, f=128, k = (self.n_bars - 1,1,1)
        , s = (1,1,1), a = 'lrelu', p = 'valid')

x = self.conv(x, f=128, k = (1,1,12), s = (1,1,12), a = 'lrelu', p = 'same') ❸
x = self.conv(x, f=128, k = (1,1,7), s = (1,1,7), a = 'lrelu', p = 'same')
x = self.conv(x, f=128, k = (1,2,1), s = (1,2,1), a = 'lrelu', p = 'same') ❹
x = self.conv(x, f=128, k = (1,2,1), s = (1,2,1), a = 'lrelu', p = 'same')
```

```
x = self.conv(x, f=256, k = (1,4,1), s = (1,2,1), a = 'lrelu', p = 'same')
x = self.conv(x, f=512, k = (1,3,1), s = (1,2,1), a = 'lrelu', p = 'same')

x = Flatten()(x)

x = Dense(1024, kernel_initializer = self.weight_init)(x)
x = LeakyReLU()(x)
critic_output = Dense(1, activation=None
        , kernel_initializer = self.weight_init)(x) ❺

self.critic = Model(critic_input, critic_output)
```

❶ На вход критика подается массив партитуры с формой `[n_bars, n_steps_per_bar, n_pitches, n_tracks]`.

❷ Сначала выполняется свертка тензора вдоль оси тактов. Для работы с четырехмерными тензорами мы используем слои `Conv3D`.

❸ Далее выполняется свертка тензора вдоль оси нот.

❹ Наконец, выполняется свертка тензора вдоль оси временных шагов.

❺ Результат формируется слоем `Dense` с единственным узлом без функции активации, как того требует механизм WGAN-GP.

## Анализ сети MuseGAN

Можете немного поэкспериментировать с вашей сетью MuseGAN, генерируя партитуры и затем настраивая некоторые параметры входного шума, чтобы увидеть, как они влияют на результат. Результатом генератора является массив значений в диапазоне [–1, 1] (из-за функции активации *tanh* в последнем слое). Чтобы преобразовать его в единственную ноту в каждой партии, для каждого временного шага выбирается нота с максимальным значением из всех 84 нот. В оригинальной статье с описанием MuseGAN авторы используют пороговое значение 0, поскольку каждая партия может содержать несколько нот, но в нашем случае с хоралами Баха можно просто взять максимальное значение, чтобы гарантировать ровно одну ноту на каждом шаге в каждой партии.

На рис. 7.16 показана партитура, сгенерированная моделью из векторов со случайным нормально распределенным шумом (вверху слева). Мы можем найти ближайшую (по евклидову расстоянию) партитуру в наборе данных и убедиться, что сгенерированная партитура не является копией

музыкальной пьесы, существующей в наборе данных, — ближайшая партитура показана чуть ниже, и, как нетрудно заметить, она не похожа на сгенерированную партитуру.

Теперь поэкспериментируем со входным шумом, чтобы улучшить сгенерированную партитуру. Для начала изменим вектор шума (см. нижнюю часть рис. 7.16). Как видим, все партии изменились, как и ожидалось, и такты теперь имеют разные свойства. Во втором такте базовая партия более динамична, а верхняя имеет более высокую тональность, чем в первом такте. При изменении стиля (вверху справа) оба такта меняются одинаково. Такты не имеют большой разницы в стиле, но весь фрагмент изменился по сравнению с первоначально сгенерированной партитурой.

**Рис. 7.16.** Партитура, сгенерированная сетью MuseGAN, ближайшая к ней реальная партитура из обучающих данных и примеры влияния попыток изменить входной шум на сгенерированную партитуру

Мы также можем изменять отдельные партии через входные тензоры мелодии и дорожек. На рис. 7.16 можно видеть эффект изменения входного шума в начальном тензоре мелодии только для верхней музыкальной партии. Все остальные части остаются неизменными, но верхняя партия изменилась довольно существенно. Можно заметить и изменение ритма между тактами в верхней партии: второй такт получился более динамичным — он содержит более короткие ноты, чем первый.

Наконец, внизу справа на рис. 7.16 показана партитура, сгенерированная после изменения входного тензора дорожек только для нижней партии. И снова все остальные части остаются неизменными, а изменилась только нижняя партия. Более того, в общем и целом такты в нижней партии сохранили сходство между собой, как и следовало ожидать.

Этот пример показывает, как, изменяя входные параметры, можно влиять непосредственно на высокоуровневые признаки сгенерированной музыкальной последовательности, — почти так же, как в предыдущих главах, где, регулируя скрытые векторы в вариационных автокодировщиках и генеративно-состязательных сетях, мы изменяли внешний вид сгенерированных образов. Один из недостатков модели — в необходимости заранее определить количество генерируемых тактов. Чтобы избавиться от него, авторы предложили расширение, реализующее передачу предыдущих тактов на вход, что позволяет модели генерировать длинные партитуры за счет непрерывной передачи самых последних сгенерированных тактов обратно в модель в качестве дополнительных входных данных.

## Итоги

Итак, мы рассмотрели два разных типа моделей для генерирования музыки: многослойную сеть LSTM с механизмом внимания и MuseGAN. Своим дизайном многослойная сеть LSTM напоминает сети для генерирования текста (см. главу 6). Музыка и текст имеют много общего, и часто для их генерирования можно использовать одинаковые методы. Мы расширили рекуррентную сеть, добавив в нее механизм внимания, позволяющий модели сосредоточиться на конкретных предыдущих временных шагах, решая задачу предсказания следующей ноты, и увидели, как модель смогла выявить такие понятия, как октавы и тональности, научившись точно имитировать музыку Баха.

Затем мы увидели, что для генерирования последовательных данных необязательно использовать рекуррентную модель — сеть MuseGAN создает по-

лифонические музыкальные партитуры с несколькими партиями, используя сверточные слои и обрабатывая партитуру как своеобразное изображение, в котором партии играют роль каналов изображения. Новизна MuseGAN заключается в том, что четыре входных тензора с шумом (аккорды, стиль, мелодия и дорожки) организованы так, чтобы с их помощью можно было управлять высокоуровневыми свойствами музыки. Да, базовая гармония все еще не так совершенна или разнообразна, как в произведениях Баха, но эта сеть является хорошей попыткой решить чрезвычайно сложную задачу и подчеркивает способность генеративно-состязательных сетей решать широкий спектр задач.

В следующей главе мы познакомимся с одной из самых замечательных моделей, разработанных в последние годы, — с *моделью мира*. Авторы этой модели в своей новаторской статье показывают, как построить модель, которая позволяет автомобилю проехать по сымитированной гоночной трассе, предварительно проверяя стратегии в своем «представлении» об окружающей среде. Это позволяет автомобилю преуспеть в движении по реальной трассе, основываясь на «тест-драйве» по воображаемой модели мира.

Глава 8

# Играем в игры

В марте 2018 года Дэвид Ха и Юрген Шмидхубер опубликовали статью «World Models».[1] В ней они показали, как обучить модель, способную выполнять определенную задачу, путем экспериментов в своих генеративных представлениях, а не в самом окружении. Это — отличный пример использования генеративного моделирования для решения практических задач, когда оно применяется наряду с другими методами машинного обучения, в том числе такими, как обучение с подкреплением.

Ключевым компонентом архитектуры является генеративная модель, способная вычислить распределение вероятностей для следующего возможного состояния с учетом текущего состояния и предполагаемого действия. Получив представление о физике окружающей среды с помощью случайных движений, модель может затем научиться решать совершенно новые задачи, действуя исключительно в рамках своего внутреннего представления об окружающей среде. Этот подход позволил получить лучшие результаты в обеих задачах, на которых он был протестирован.

В этой главе мы подробно исследуем модель и посмотрим, как создать свою версию этой удивительной передовой технологии. Опираясь на статью (см. выше), построим алгоритм обучения с подкреплением, который научится водить машину на гоночной трассе. В качестве окружающей среды используем двумерную компьютерную модель, но это не значит, что данный метод нельзя применить к сценариям реального мира, когда тестирование в реальной обстановке стоит слишком дорого или в принципе невозможно.

[1] David Ha и Jürgen Schmidhuber, «World Models», 27 марта 2018, https://arxiv.org/abs/1803.10122.

Но прежде чем приступить к созданию модели, рассмотрим более подробно идею обучения с подкреплением и платформу OpenAI Gym.

## Обучение с подкреплением

Обучение с подкреплением можно определить так:

> Обучение с подкреплением (Reinforcement Learning, RL) — область машинного обучения, целью которой является обучение агента для оптимального решения конкретных задач в конкретной среде.

В отличие от дискриминативного и генеративного моделирования, направленных на минимизацию функции потерь по набору наблюдений, обучение с подкреплением стремится максимизировать долгосрочное вознаграждение агента в данном окружении. Этот подход часто описывается как одна из трех основных ветвей машинного обучения наряду с *обучением с учителем* (прогнозирование с использованием маркированных данных) и *обучением без учителя* (обучение по немаркированным данным). Для начала познакомимся с некоторыми ключевыми терминами, относящимися к обучению с подкреплением:

*Окружение (среда)*

Мир, в котором действует агент. Характеристики мира определяются набором правил, которые управляют процессом обновления состояния игры и распределением вознаграждений, с учетом предыдущих действий агента и текущего состояния игры. Например, для алгоритма обучения с подкреплением, изучающего игру в шахматы, среда будет характеризоваться правилами, определяющими, как данное действие (например, ход `e2-e4`) влияет на следующее состояние игры (новое расположение фигур на доске) и как определить, является ли данная позиция матом, и назначить победившему игроку вознаграждение 1 после победного хода.

*Агент*

Сущность, предпринимающая действия в окружающей, реально складывающейся или смоделированной обстановке.

*Состояние игры*

Данные, описывающие конкретную ситуацию, с которой может столкнуться агент (иногда называется просто *состоянием*), например конкретное расположение фигур на шахматной доске с дополнительной игровой информацией (например, о том, кому принадлежит право следующего хода).

*Действие*

Ход, который может сделать агент.

*Вознаграждение*

Ценность, возвращаемая агенту средой после выполнения действия. Агент стремится максимизировать долгосрочную сумму своих вознаграждений. Например, в шахматах мат королю противника позволяет получить вознаграждение 1, а за каждый следующий ход вознаграждение равно 0. В других играх вознаграждения постоянно присуждаются на протяжении всего эпизода (например, очки в игре *Космические захватчики*).

*Эпизод*

Один сеанс действий агента в окружении; иногда эпизод называют также *прогоном*.

*Временной шаг*

В дискретном окружении все состояния, действия и вознаграждения относятся к конкретному временному шагу $t$ (рис. 8.1).

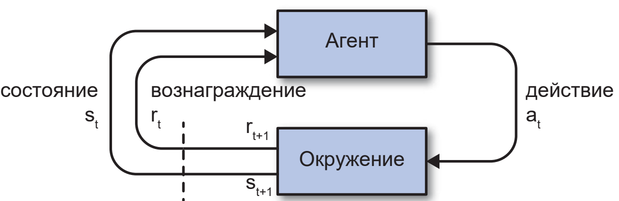



**Рис. 8.1.** Диаграмма работы алгоритма обучения с подкреплением

Среда инициализируется с текущим состоянием игры, $s_0$. На временном шаге $t$ агент получает текущее состояние игры $s_t$ и использует его для выбора следующего оптимального действия, которое затем выполняет. После действия агента среда вычисляет следующее состояние $s_{t+1}$ и величину вознаграждения $r_{t+1}$, возвращая их агенту, чтобы тот мог начать новый цикл. Цикл продолжается до тех пор, пока не будет достигнут критерий окончания эпизода (например, заданное число временных шагов или победа/поражение агента).

Как заставить агента максимизировать сумму вознаграждений в данном окружении? Можно определить набор правил, руководствуясь которыми агент должен реагировать на то или иное состояние игры. Однако этот подход быстро становится неосуществимым по мере усложнения среды — мы не сможем создать агента, обладающего сверхчеловеческими способностями в конкретной задаче, жестко задавая правила. Обучение с подкреплением предусматривает создание агента, который может самостоятельно выявлять оптимальные стратегии в сложных условиях, многократно повторяя игру, — именно этот подход мы и используем в этой главе, чтобы создать своего агента. А теперь познакомимся с OpenAI Gym, домом для среды `CarRacing`, которую мы используем для имитации движения по гоночной трассе.

## OpenAI Gym

OpenAI Gym (https://gym.openai.com/) — это набор инструментов для разработки алгоритмов обучения с подкреплением, доступный в виде библиотеки для Python.

В состав библиотеки входит несколько классических окружений для обучения с подкреплением, таких как `CartPole` и `Pong`, а также окружения, представляющие более сложные задачи, такие как обучение агента ходьбе по неровной местности или выигрышу в игре Atari. Все окружения предлагают метод `step`, вызовом которого можно сообщить о выполненном действии; в ответ окружение вернет следующее состояние и вознагражде-

ние. Многократно вызывая метод `step` с действиями, выбранными агентом, можно воспроизвести эпизод в окружении. В дополнение к абстрактной механике каждого окружения OpenAI Gym также предлагает графические средства, позволяющие наблюдать за работой агента в данном окружении. Это может пригодиться при отладке и поиске направлений для дальнейшего совершенствования агента. Далее мы будем использовать окружение `CarRacing` из OpenAI Gym. Посмотрим, как в нем определяются состояние игры, действия, вознаграждение и эпизод:

*Состояние игры*

Изображение $64 \times 64$ пиксела в формате RGB, показывающее вид сверху на трассу и автомобиль.

*Действие*

Набор из трех значений: направление (угол поворота руля (от $-1$ до 1)), ускорение (от 0 до 1) и торможение (от 0 до 1). На каждом временном шаге агент должен установить все три значения.

*Вознаграждение*

Отрицательный штраф $-0.1$ за каждый временной шаг и положительное вознаграждение $1000/N$ за пересечение границы нового фрагмента трека, где $N$ — общее количество фрагментов, составляющих трек.

*Эпизод*

Эпизод завершается, когда автомобиль пересекает финишную черту или по прошествии 3000 временных шагов.

На рис. 8.2 изображено графическое представление игры. Обратите внимание: агент видит трек как бы с высоты птичьего полета и управляет автомобилем, который движется по треку.

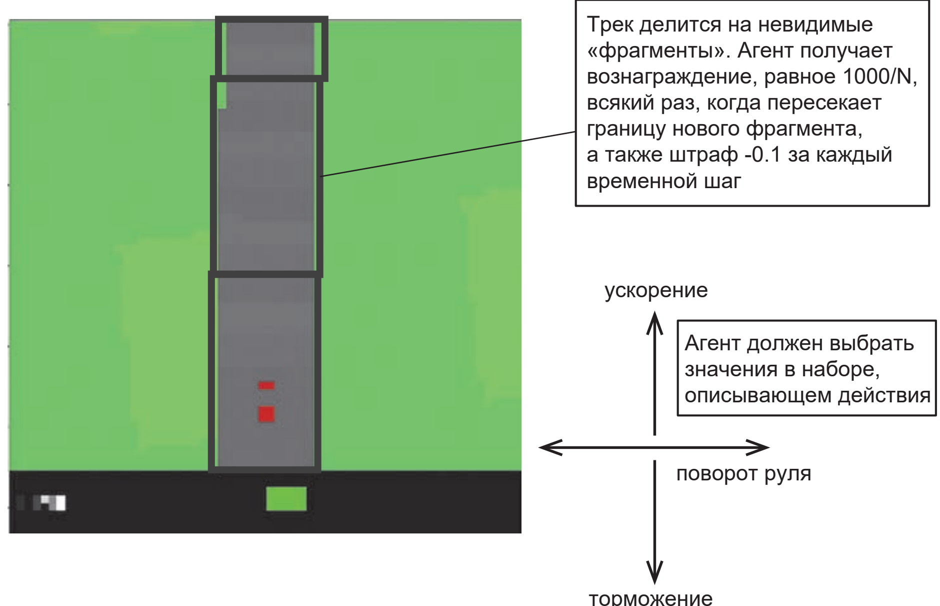



**Рис. 8.2.** Графическое представление одного состояния игры в окружении CarRacing

## Архитектура модели мира

Теперь рассмотрим в общих чертах архитектуру, которую будем использовать для создания агента, обучающегося методом обучения с подкреплением, а потом перейдем к подробному описанию этапов создания каждого компонента. Решение состоит из трех основных компонентов, изображенных на рис. 8.3, обучаемых независимо:

*V*

Вариационный автокодировщик (VAE).

*M*

Рекуррентная нейронная сеть с полносвязанной сетью смеси распределений (MDN-RNN).

*C*

Контроллер.

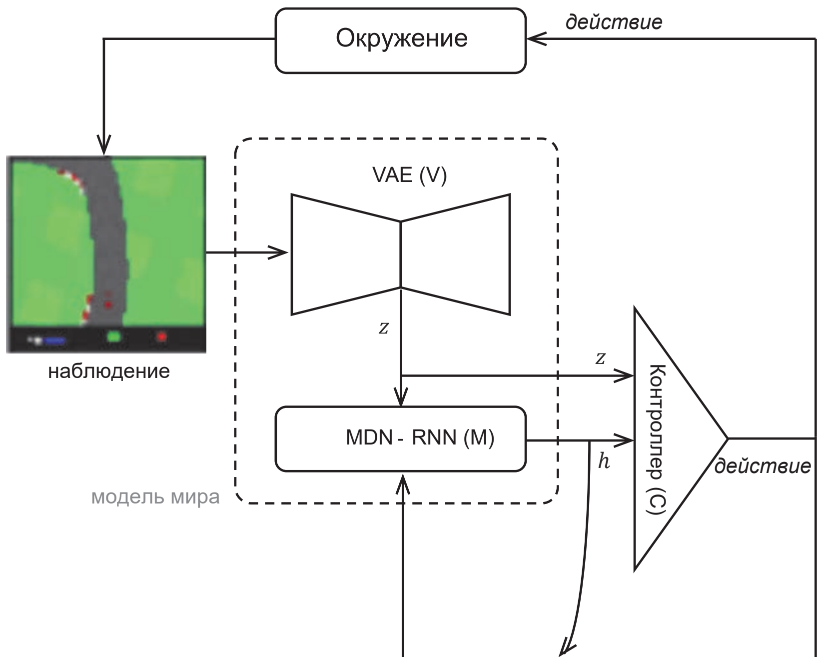



**Рис. 8.3.** Архитектура модели мира

## Вариационный автокодировщик

При принятии решений во время вождения никто из нас не анализирует каждый видимый пиксел — вместо этого мы уплотняем визуальную информацию по нескольким параметрам: прямолинейность дороги, предстоящие повороты и положение автомобиля на дороге, что помогает выбрать следующее действие. В главе 3 мы видели, как вариационный автокодировщик принимает на входе изображение большого размера и путем минимизации ошибки восстановления и расстояния Кульбака—Лейблера (Kullback—Leibler, KL) сжимает его в некоторую случайную переменную в скрытом пространстве, приблизительно соответствующую стандартному многомерному нормальному распределению. Это гарантирует непрерывность скрытого пространства и возможность легко делать выборки из него, чтобы генерировать новые значимые наблюдения.

В примере автомобильных гонок вариационный автокодировщик сжимает входное изображение 64 × 64 × 3 (RGB) в 32-мерную нормально распреде-

ленную случайную величину, параметризованную двумя переменными, `mu` и `log_var`, где `log_var` — это логарифм дисперсии распределения. Мы можем сделать выборку из распределения, чтобы получить скрытый вектор `z`, представляющий текущее состояние. Этот вектор передается следующему компоненту сети MDN-RNN.

## Сеть MDN-RNN

Во время вождения каждое следующее наблюдение не является для нас полной неожиданностью. Если текущее наблюдение показывает впереди левый поворот, то, поворачивая руль влево, мы ожидаем, что к моменту следующего наблюдения все еще будем находиться на дороге. Если бы это было не так, мы ездили бы по дорогам, виляя из стороны в сторону, так как не замечали бы небольшого отклонения от центра на следующем временном шаге, и затем оказываясь в положении, требующем немедленных и активных действий.

Эта предусмотрительность — зона ответственности сети MDN-RNN, которая пытается предсказать распределение следующего скрытого состояния на основе предыдущего скрытого состояния и предыдущего действия. В частности, MDN-RNN — это слой LSTM с 256 скрытыми узлами, за которым следует выходной слой сети смеси распределений (Mixture Density Network, MDN), который учитывает тот факт, что следующее скрытое состояние фактически можно получить из любого из нескольких нормальных распределений.

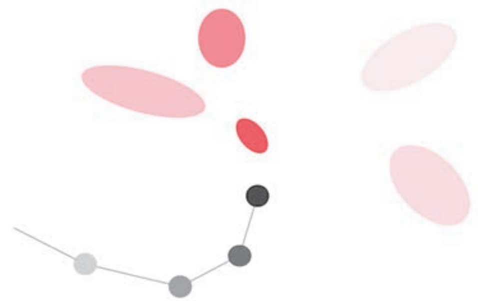


**Рис. 8.4.** Генерация рукописного текста с помощью MDN

Тот же метод применялся и одним из авторов статьи «World Models» Дэвидом Ха, к задаче генерации рукописного текста (http://bit.ly/2Wm9X01), чтобы описать, что далее перо может двигаться в сторону любой из красных областей (рис. 8.4).

В примере с гоночным автомобилем мы допускаем, что каждый элемент следующего наблюдаемого скрытого состояния может быть получен из любого из пяти нормальных распределений.

## Контроллер

До этого момента мы вообще не упоминали выбор действия. Эта ответственность лежит на контроллере, реализованном как полносвязанная нейронная сеть, на вход которой подается конкатенация вектора `z` (текущего скрытого состояния, выбранного из распределения, закодированного вариационным автокодировщиком) и скрытого состояния сети RNN. Три выходных нейрона соответствуют трем действиям (поворот, ускорение, торможение) и масштабируются, чтобы попасть в соответствующие диапазоны.

Мы должны обучить контроллер, используя метод обучения с подкреплением, потому что у нас нет обучающего набора данных, который помогал бы понять, какое действие *хорошее*, а какое *плохое*. Агент должен сам определить это путем повторных экспериментов. Как мы увидим далее, суть статьи «World Models» заключается в демонстрации, как обучение с подкреплением может происходить внутри генеративной модели окружения в агенте, а не в окружении OpenAI Gym. Иными словами, это происходит не в реальном окружении, а в его версии, *вымышленной* агентом. Чтобы понять роль каждого из этих трех компонентов и то, как они работают вместе, вообразим себе следующий диалог:

> *VAE* (просматривая последнее наблюдение 64 × 64 × 3): выглядит как прямая дорога, с пологим левым поворотом, к которому приближается автомобиль, следующий в направлении дороги (`z`).
>
> *RNN:* исходя из этого описания (`z`) и того факта, что на предыдущем шаге контроллер решил сильно ускориться (`action`), я обновлю свое скрытое состояние и в следующем наблюдении дорога по-прежнему останется прямой дорогой, но с немного большим левым поворотом впереди.
>
> *Контроллер:* исходя из описания, полученного от VAE (`z`) и текущего скрытого состояния RNN (`h`), моя нейронная сеть предсказывает следующее действие `[0.34, 0.8, 0]`.

Затем действие, сгенерированное контроллером, передается в окружение, возвращающее обновленное наблюдение, и цикл начинается снова. Дополнительную информацию о модели можно получить здесь: https://worldmodels.github.io/.

## Подготовка

Теперь можно приступать к созданию и обучению этой модели с помощью Keras. Если у вас есть ноутбук с высокими техническими характеристиками, можете запустить решение на нем, но я бы порекомендовал использовать облачные ресурсы вроде Google Cloud Compute Engine (https://cloud.google.com/compute), получив доступ к мощным компьютерам, которыми можно пользоваться, запуская короткие сеансы.

Следующий код протестирован в Ubuntu 16.04, поэтому его следует выполнять в терминале Linux.

Сначала установим следующие библиотеки:

```
sudo apt-get install cmake swig python3-dev \
        zlib1g-dev python-opengl mpich xvfb \
        xserver-xephyr vnc4server
```

Затем клонируем следующий репозиторий:

```
git clone https://github.com/AppliedDataSciencePartners/WorldModels.git
```

Так как код проекта хранится отдельно от репозитория книги, предлагаю создать отдельное виртуальное окружение для экспериментов:

```
mkvirtualenv worldmodels
cd WorldModels
pip install -r requirements.txt
```

Теперь можно двигаться вперед!

## Обзор процесса обучения

Процесс обучения включает пять этапов:

1. Сбор данных в ходе случайных прогонов. Здесь перед агентом не стоит цель решить задачу — он просто исследует окружение случайным образом. На этом этапе с помощью OpenAI Gym имитируется несколько эпизодов, и на каждом временном шаге сохраняются наблюдаемое состояние, действие и вознаграждение. Идея в том, чтобы получить данные о работе окружения, которые затем VAE сможет изучить и научиться представлять состояния в виде векторов в скрытом пространстве. Затем MDN-RNN сможет узнать, как скрытые векторы развиваются с течением времени.

2. Обучение VAE. Используя данные, собранные в ходе случайных прогонов, обучим VAE на наблюдаемых изображениях.

3. Сбор данных для обучения MDN-RNN. После обучения VAE используем его для кодирования каждого из собранных наблюдений в векторы `mu` и `log_var`, сохранив их вместе с текущим действием и вознаграждением.

4. Обучение MDN-RNN. Возьмем пакеты по 100 эпизодов и на каждом временном шаге загрузим переменные `mu`, `log_var`, `action` и `reward`, которые были сгенерированы на шаге 3. Затем выберем вектор `z` из векторов `mu` и `log_var`. Затем на текущем векторе `z`, `action` и `reward` обучим MDN-RNN предсказывать следующий вектор `z` и `reward`.

5. Обучение контроллера. Используя подготовленные VAE и RNN, обучим контроллер предсказывать действие для текущего `z` и скрытого состояния `h` сети RNN. В качестве оптимизатора контроллер использует эволюционный алгоритм CMA-ES (Covariance Matrix Adaptation Evolution Strategy — эволюционная стратегия с адаптацией матрицы ковариаций). Он поощряет веса матрицы, которые генерируют действия, приводящие к более высокому общему вознаграждению, поэтому такое желаемое поведение почти наверняка будет унаследовано будущими поколениями.

А теперь подробно рассмотрим каждый этап.

## Сбор данных в ходе случайных прогонов

Чтобы запустить сбор данных, выполните следующую команду в терминале:

```
bash 01_generate_data.sh <env_name> <parallel_process> <episodes_per_process> \
<render> <action_refresh_rate>
```

передав ей следующие параметры:

*<env_name>*

Имя окружения, используемого функцией make_env (например, car_racing).

*<parallel_process>*

Число одновременно запускаемых процессов для обработки (например, 8 для компьютера с 8-ядерным процессором).

*<episodes_per_process>*

Сколько эпизодов должен запустить каждый процесс (например, 125, чтобы 8 процессов в общей сложности создали 1000 эпизодов).

*<max_timesteps>*

Максимальное число временных шагов в одном эпизоде (например, 300).

*<render>*

1 — для отображения процесса прогона в окне (иначе 0).

*<action_refresh_rate>*

Количество временных шагов, для которых необходимо зафиксировать текущее действие перед изменением. Это препятствует слишком быстрому изменению действий в процессе движения автомобиля.

Например, для компьютера с 8-ядерным процессором команда могла бы выглядеть так:

```
bash 01_generate_data.sh car_racing 8 125 300 0 5
```

Она запустит 8 параллельных процессов, каждый из которых сымитирует 125 эпизодов с максимальной длительностью 300 временных шагов каждый и обновлением действия через каждые пять временных шагов. Каждый процесс будет выполнять сценарий на Python 01_generate_data.py. Ключевая часть сценария представлена в листинге 8.1.

**Листинг 8.1.** Выдержка из сценария 01_generate_data.py

```
# ...

DIR_NAME = './data/rollout/'

env = make_env(current_env_name) ❶
s = 0
while s < total_episodes:
    episode_id = random.randint(0, 2**31-1)
    filename = DIR_NAME + str(episode_id)+".npz"
    observation = env.reset()
    env.render()
    t = 0
    obs_sequence = []
    action_sequence = []
    reward_sequence = []
    done_sequence = []
    reward = -0.1
    done = False

    while t < time_steps:
        if t  % action_refresh_rate == 0:
            action = config.generate_data_action(t, env) ❷
        observation = config.adjust_obs(observation) ❸
        obs_sequence.append(observation)
        action_sequence.append(action)
        reward_sequence.append(reward)
        done_sequence.append(done)
        observation, reward, done, info = env.step(action) ❹
        t = t + 1

    print("Episode {} finished after {} timesteps".format(s, t))
    np.savez_compressed(filename
        , obs=obs_sequence
        , action=action_sequence
        , reward = reward_sequence
        , done = done_sequence) ❺
    s = s + 1
env.close()
```

❶ `make_env` — пользовательская функция из репозитория, которая создает соответствующее окружение OpenAI Gym. В данном случае мы создаем среду `CarRacing` с несколькими настройками. Файл окружения хранится в папке *custom_envs*.

❷ `generate_data_action` — пользовательская функция, которая определяет правила для выбора случайных действий.

❸ Наблюдения, возвращаемые окружением, содержат числа в диапазоне от 0 до 255. Нам нужно масштабировать их в диапазон от 0 до 1, поэтому данная функция просто делит числа на 255.

❹ Каждое окружение в OpenAI Gym включает метод `step`. Для заданного действия он возвращает следующее наблюдение, вознаграждение и флаг «выполнено».

❺ Каждый эпизод мы сохраняем в отдельном файле в каталоге *./data/rollout/*.

На рис. 8.5 показаны кадры 40–59 из одного эпизода, когда машина приближается к повороту, а также случайно выбранные действия и вознаграждения. Обратите внимание, что вознаграждение увеличивается до 3,22, когда машина пересекает границу нового фрагмента трека, а в остальных случаях взыскивается штраф –0,1. Кроме того, действие меняется через каждые пять кадров, потому что параметр `action_refresh_rate` равен `5`.

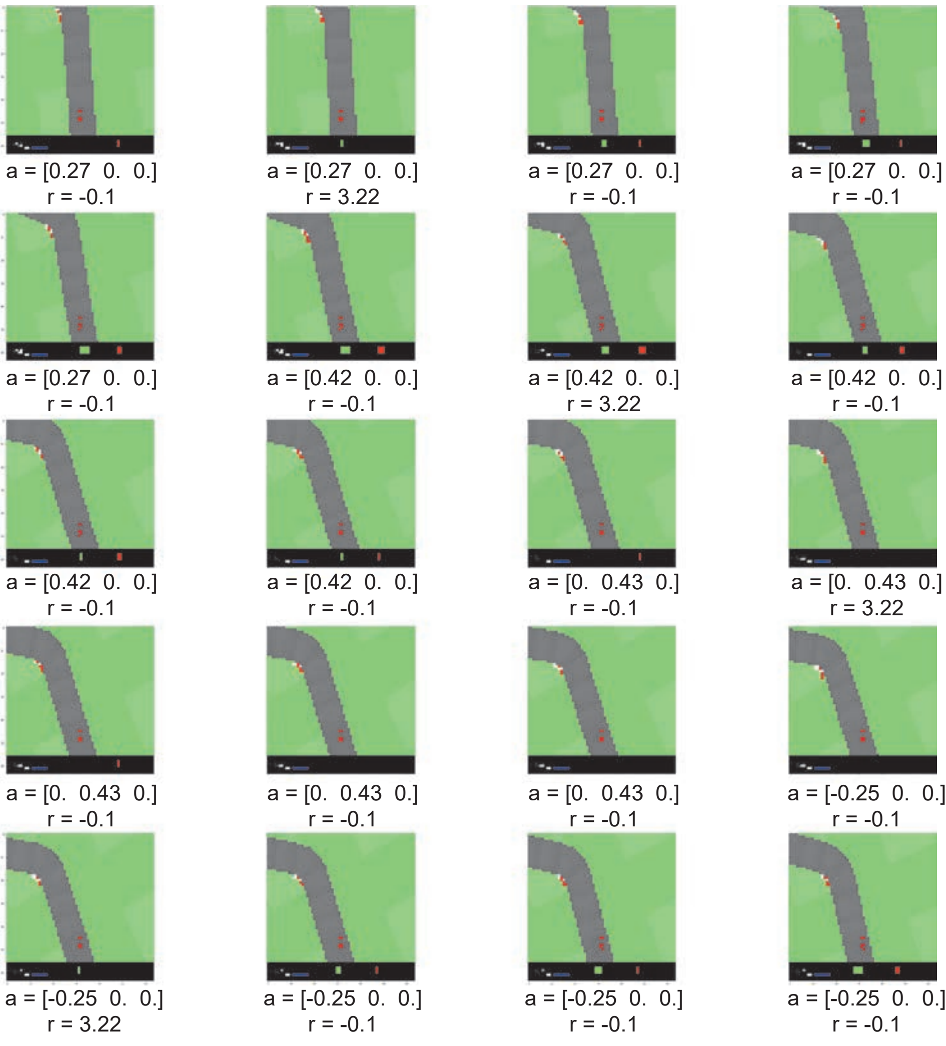



**Рис. 8.5.** Кадры 40–59 из одного эпизода

## Обучение VAE

Теперь на основе собранных данных можно построить генеративную модель (вариационный автокодировщик VAE). Напомним, цель VAE — дать возможность свернуть одно изображение размером 64 × 64 × 3 в нормально распределенную случайную величину, распределение которой параметризовано двумя векторами, `mu` и `log_var`. Каждый из этих векторов имеет длину 32.

Чтобы запустить обучение VAE, выполните следующую команду в терминале:

```
python 02_train_vae.py --new_model [--N] [--epochs]
```

передав ей следующие параметры:

`--new_model`

: Ключ, требующий обучить модель с нуля. Используйте этот флаг при первом обучении; если его опустить, то код будет искать файл ./vae/vae.json и продолжать обучение предыдущей модели.

`--N` (*необязательный*)

: Количество эпизодов, используемых при обучении VAE (например, `1000` — для достижения хороших результатов необязательно использовать все эпизоды, поэтому, чтобы ускорить обучение, можно использовать только часть эпизодов).

`--epochs` (*необязательный*)

: Число эпох обучения (например, `3`).

На рис. 8.6 показано, как должен выглядеть процесс обучения в терминале. Веса обученной сети сохраняются в конце каждой эпохи в файле ./vae/vae.json.

```
Imported 850 / 1000 ::: Current data size = 255000 observations
Imported 900 / 1000 ::: Current data size = 270000 observations
Imported 950 / 1000 ::: Current data size = 285000 observations
Imported 1000 / 1000 ::: Current data size = 300000 observations
Imported 1000 / 1000 ::: Current data size = 300000 observations
DATA SHAPE = (300000, 64, 64, 3)
EPOCH 0
Epoch 1/1
300000/300000 [==============================] - 2632s 9ms/step - loss: 86.7616 - vae_r_loss: 69.8272 - vae_kl_loss: 16.9344
EPOCH 1
Epoch 1/1
300000/300000 [==============================] - 5270s 18ms/step - loss: 48.5049 - vae_r_loss: 32.0335 - vae_kl_loss: 16.4714
EPOCH 2
Epoch 1/1
300000/300000 [==============================] - 5151s 17ms/step - loss: 43.7458 - vae_r_loss: 27.3376 - vae_kl_loss: 16.4081
EPOCH 3
Epoch 1/1
300000/300000 [==============================] - 3423s 11ms/step - loss: 41.6400 - vae_r_loss: 25.2663 - vae_kl_loss: 16.3737
```

**Рис. 8.6.** Обучение VAE

## Архитектура VAE

Функциональный программный интерфейс библиотеки Keras позволяет определить не только полную модель VAE для обучения, но и дополнительные модели, представляющие отдельно кодировщика и декодировщика обучаемой сети. Они могут пригодиться, если вдруг появится желание закодировать конкретное изображение или декодировать некоторый вектор `z`. В этом примере мы определим четыре разные модели внутри VAE:

`full_model`

Полная модель для обучения.

`encoder`

Принимает наблюдение 64 × 64 × 3 и выводит вектор `z`. Если вызвать метод `predict` этой модели для одного и того же наблюдения несколько раз, то он вернет разные результаты, потому что, несмотря на постоянство значений `mu` и `log_var`, случайный вектор `z` каждый раз будет отличаться.

`encoder_mu_log_var`

Принимает наблюдение 64 × 64 × 3 и выводит векторы `mu` и `log_var`, соответствующие этому наблюдению. В отличие от модели `encoder`, если вызвать метод `predict` этой модели несколько раз, каждый раз он будет возвращать один и тот же результат: вектор `mu` с длиной 32 и вектор `log_var` с длиной 32.

`decoder`

Принимает вектор `z` и возвращает восстановленное наблюдение 64 × 64 × 3.

На рис. 8.7 представлена диаграмма, отражающая архитектуру VAE. Вы можете поэкспериментировать с этой архитектурой, отредактировав файл ./vae/arch.py, где определяются класс вариационного автокодировщика и параметры нейронной сети.

## Анализ VAE

Теперь посмотрим, что возвращают методы `predict` различных моделей в VAE, чтобы понять, чем они отличаются, а затем — то, как VAE может создавать совершенно новые наблюдения трека.

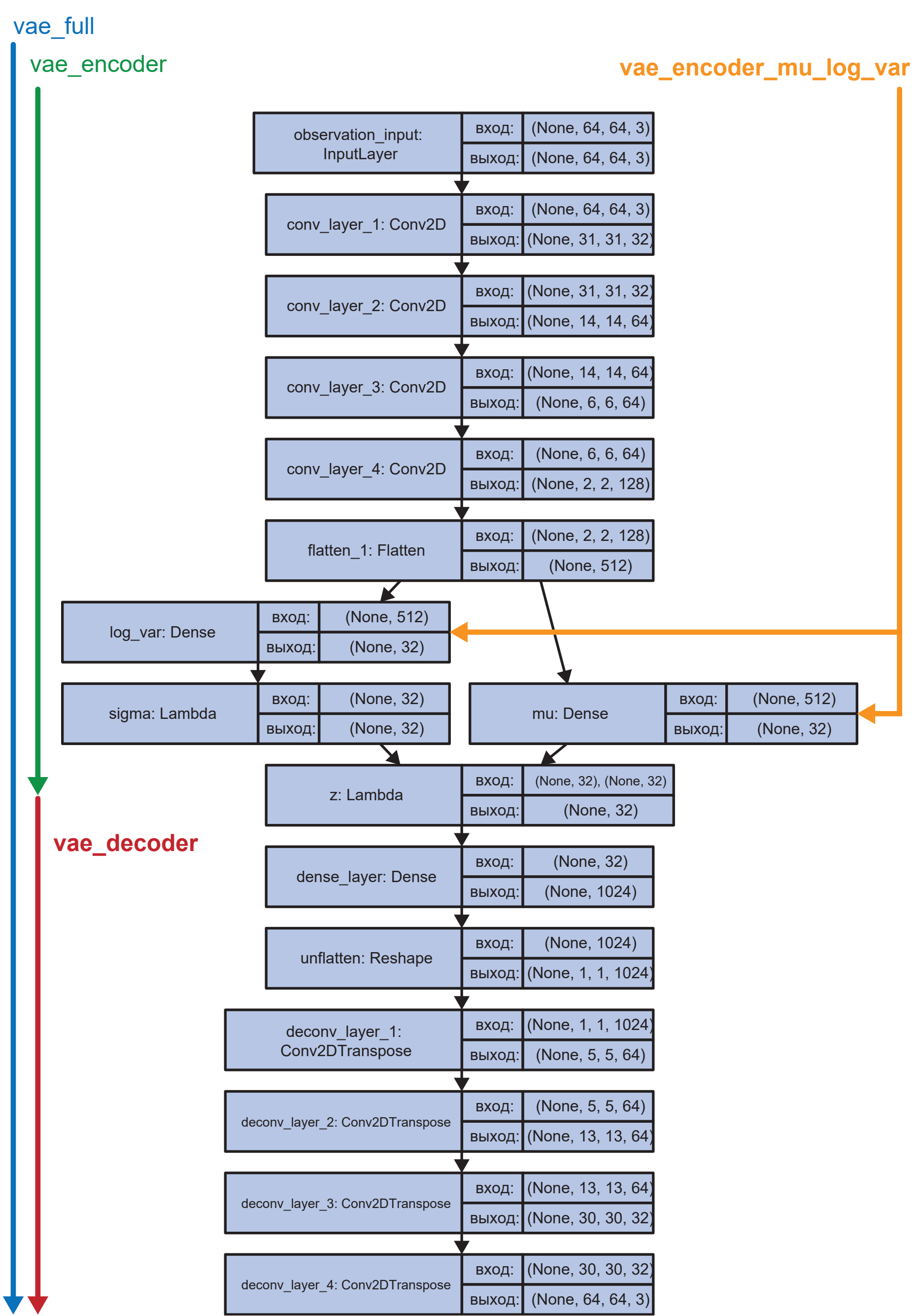



**Рис. 8.7.** Архитектура вариационного автокодировщика из статьи «World Models»

## Полная модель

После обучения модели `full_model` на наблюдениях она сможет восстановить точное изображение трека (рис. 8.8). Это может пригодиться для визуальной проверки правильности работы VAE.

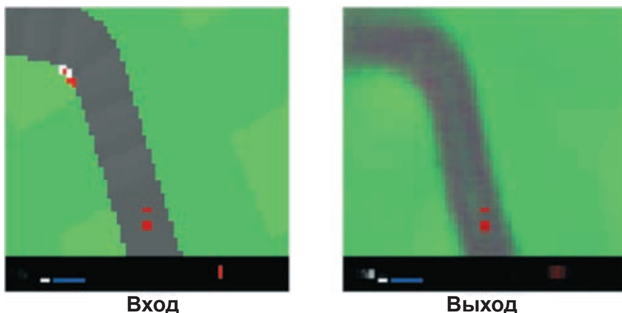



**Рис. 8.8.** Входные данные полной модели VAE и полученный ею результат

## Модели кодировщиков

После обучения модели `encoder_mu_log_var` на наблюдениях она будет возвращать сгенерированные векторы `mu` и `log_var`, описывающие многомерное нормальное распределение. Модель `encoder` делает еще один шаг вперед и выбирает конкретный вектор `z` их этого распределения. Результаты, возвращаемые этими моделями кодировщиков, показаны на рис. 8.9.

```
encoded_mu_log_var = vae.encoder_mu_log_var.predict(np.array([obs]))
mu = encoded_mu_log_var[0][0]
log_var = encoded_mu_log_var[1][0]
print("mu = " + str(mu))
print("log_var = " + str(log_var))

encoded_z = vae.encoder.predict(np.array([obs]))[0]
print("z = " + str(encoded_z))
```

```
mu = [ 0.461   0.2408  0.9884  0.0459  0.9623  1.0147  0.6555 -0.586  -0.4304
 -0.0892 -1.5475  0.0319  0.426   0.1936  0.0838 -0.9127 -0.0823 -0.0321
  0.0829 -0.2372 -0.1084 -0.0069  0.011  -0.0201 -1.735  -0.8933  0.0082
  0.5342  0.1982  0.056  -0.0389  0.0389]
log_var = [-1.8887 -0.1232 -2.1576  0.0093 -5.2961 -2.2058 -3.0662 -3.1458  0.0018
 -0.0412 -3.1433  0.0278 -0.0169  0.0186 -0.0313 -0.8884 -0.0468 -0.0565
  0.0075  0.0149 -0.0285 -0.0013 -0.006  -0.0468 -3.2442 -3.9082  0.0348
 -2.0272 -0.0264 -0.0202 -0.0585 -0.0103]
z = [ 0.4681  0.3364  0.969  -0.3118  1.0631  1.1805  1.1039 -0.6603 -1.5317
 -0.0679 -1.4819 -0.537   1.2687 -1.2981 -0.0219 -0.7657  0.5345 -1.3642
 -0.1958 -1.2382  1.6123 -1.9301  0.1032 -0.1125 -1.8621 -1.2199  0.3788
  0.6171  0.6412 -0.6444 -0.0741 -1.4384]
```

**Рис. 8.9.** Результаты, возвращаемые моделями кодировщиков

```
plt.plot(mu);
plt.plot(log_var);
hot_zs = np.where(abs(log_var) > 0.1)[0]
hot_zs
```

```
array([ 0,  1,  2,  4,  5,  6,  7, 10, 15, 24, 25, 27])
```

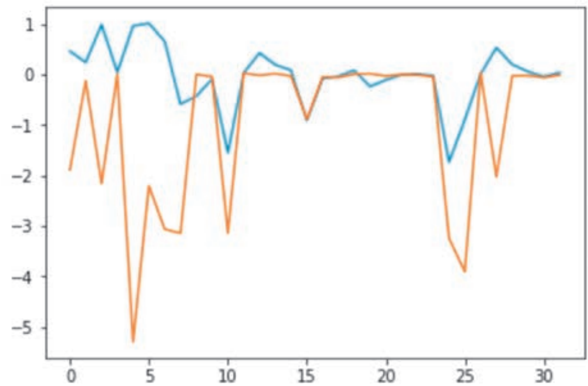



**Рис. 8.10.** График изменения значений всех 32 измерений в mu (синяя линия) и log_var (оранжевая линия) для конкретного наблюдения

Взгляните на график изменения значений всех 32 измерений в `mu` и `log_var` (рис. 8.10) для конкретного наблюдения и обратите внимание, что только 12 измерений значительно отличаются от стандартного нормального распределения (`mu = 0`, `log_var = 0`). Это связано с тем, что вариационный автокодировщик пытается минимизировать расстояние *KL*, чтобы как можно меньше измерений отличалось от стандартного нормального распределения, решив, что 12 измерений содержат достаточный объем информации о наблюдениях для точной реконструкции.

## Модель декодировщика

Модель декодировщика принимает вектор `z` и реконструирует исходное изображение. На рис. 8.11 показан результат линейной интерполяции двух измерений в `z`, позволяющий увидеть, как каждое измерение кодирует определенный аспект трека. Например, `z[4]` определяет направление трека влево/вправо относительно автомобиля, а `z[7]` — крутизну приближающегося левого поворота.

Это доказывает, что скрытое пространство, изученное вариационным автокодировщиком, является непрерывным и может использоваться для генерирования новых фрагментов трека, которые раньше не наблюдались агентом.

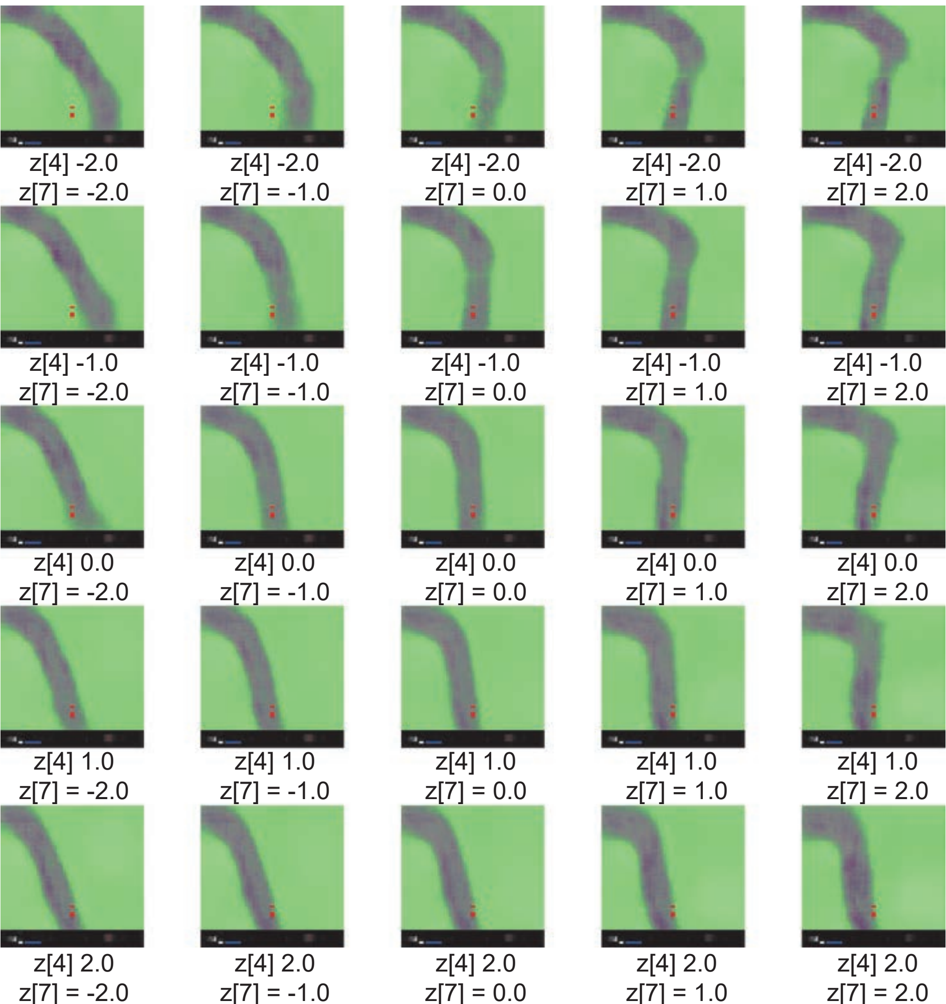



**Рис. 8.11.** Линейная интерполяция двух измерений в z

## Сбор данных для обучения RNN

Обученный VAE теперь можно использовать для генерирования обучающих данных для сети RNN.

На этом этапе пропустим все данные случайного прогона через модель `encoder_mu_log_var` и сохраним векторы `mu` и `log_var`, соответствующие каждому наблюдению. Эти закодированные данные, вместе с собранными данными о действиях и вознаграждениях, будут использоваться для обучения MDN-RNN. Запустим сбор данных, выполнив следующую команду в терминале:

```
python 03_generate_rnn_data.py
```

В листинге 8.2 приводится выдержка из файла 03_generate_data.py, иллюстрирующая, как производится генерирование данных для обучения MDN-RNN.

**Листинг 8.2.** Выдержка из файла 03_generate_data.py

```
def encode_episode(vae, episode):

    obs = episode['obs']
    action = episode['action']
    reward = episode['reward']
    done = episode['done']

    mu, log_var = vae.encoder_mu_log_var.predict(obs) ❶

    done = done.astype(int)
    reward = np.where(reward>0, 1, 0) * np.where(done==0, 1, 0) ❷

    initial_mu = mu[0, :]
    initial_log_var = log_var[0, :]

    return (mu, log_var, action, reward, done, initial_mu, initial_log_var)

vae = VAE()
vae.set_weights('./vae/weights.h5')

for file in filelist:
    rollout_data = np.load(ROLLOUT_DIR_NAME + file)
    mu, log_var, action, reward, done, initial_mu
        , initial_log_var = encode_episode(vae, rollout_data)

    np.savez_compressed(SERIES_DIR_NAME + file, mu=mu, log_var=log_var
        , action = action, reward = reward, done = done)
    initial_mus.append(initial_mu)
```

```
    initial_log_vars.append(initial_log_var)

np.savez_compressed(ROOT_DIR_NAME + 'initial_z.npz', initial_mu=initial_mus
    , initial_log_var=initial_log_vars) ❸
```

❶ Здесь используется модель `encoder_mu_log_var`, чтобы получить векторы `mu` и `log_var` для конкретного наблюдения.

❷ Значение вознаграждения преобразуется в 0 или 1, чтобы его можно было передать на вход в MDN-RNN.

❸ Начальные векторы `mu` и `log_var` сохраняются в отдельный файл — они пригодятся нам позже для инициализации воображаемого окружения.

## Обучение сети MDN-RNN

Теперь обучим сеть MDN-RNN предсказывать распределение следующего вектора `z` и вознаграждения, исходя из текущего значения `z`, текущего действия и предыдущего вознаграждения. Цель MDN-RNN — предсказать будущее на один шаг вперед. После этого мы сможем использовать внутреннее скрытое состояние LSTM для передачи на вход контроллера. Запустим обучение сети MDN-RNN, выполнив следующую команду в терминале:

```
python 04_train_rnn.py (--new_model) (--batch_size) (--steps)
```

со следующими параметрами:

`new_model`

Ключ, требующий обучить модель с нуля. Используйте этот флаг при первом обучении; если его опустить, то код будет искать файл ./rnn/rnn.json и продолжать обучение предыдущей модели.

`batch_size`

Количество эпизодов, используемых при обучении MDN-RNN в каждой итерации.

`steps`

Общее число итераций обучения.

На рис. 8.12 показано, как выглядит процесс обучения в терминале. Веса обученной сети сохраняются в файл ./rnn/rnn.json через каждые 10 шагов.

```
STEP 0
Epoch 1/1
100/100 [==============================] - 2s 20ms/step - loss: 2.0871 - rnn_z_loss: 1.3943 - rnn_rew_loss: 0.6928
STEP 1
Epoch 1/1
100/100 [==============================] - 1s 12ms/step - loss: 2.0435 - rnn_z_loss: 1.3928 - rnn_rew_loss: 0.6507
STEP 2
Epoch 1/1
100/100 [==============================] - 1s 12ms/step - loss: 2.0018 - rnn_z_loss: 1.3881 - rnn_rew_loss: 0.6138
STEP 3
Epoch 1/1
100/100 [==============================] - 1s 12ms/step - loss: 1.9670 - rnn_z_loss: 1.3881 - rnn_rew_loss: 0.5789
STEP 4
Epoch 1/1
100/100 [==============================] - 1s 12ms/step - loss: 1.9073 - rnn_z_loss: 1.3794 - rnn_rew_loss: 0.5278
STEP 5
Epoch 1/1
100/100 [==============================] - 1s 11ms/step - loss: 1.8410 - rnn_z_loss: 1.3711 - rnn_rew_loss: 0.4700
STEP 6
Epoch 1/1
100/100 [==============================] - 1s 11ms/step - loss: 1.7674 - rnn_z_loss: 1.3606 - rnn_rew_loss: 0.4068
```

**Рис. 8.12.** Обучение сети MDN-RNN

## Архитектура сети MDN-RNN

На рис. 8.13 представлена диаграмма, отражающая архитектуру сети MDN-RNN.

Сеть состоит из слоя LSTM (RNN), за которым следует полносвязанный слой (MDN), преобразующий скрытое состояние LSTM в параметры смеси распределений. Давайте рассмотрим эту архитектуру детальнее. На вход слоя LSTM подается вектор с длиной 36 — конкатенация вектора z (с длиной 32), полученного вариационным автокодировщиком, текущего действия (с длиной 3) и предыдущего вознаграждения (с длиной 1). На выходе слой LSTM возвращает вектор с длиной 256 — по одному значению для каждой ячейки в слое LSTM. Этот вектор передается в MDN — обычный полносвязанный слой, который преобразует вектор с длиной 256 в вектор с длиной 481.

Почему 481? На рис. 8.14 показано, как устроен вектор, получаемый на выходе из MDN-RNN. Напомним, цель сети смеси распределений — смоделировать возможность получения следующего вектора состояния z из

нескольких возможных распределений с определенной вероятностью. В примере с гоночным автомобилем мы выбрали пять нормальных распределений. Сколько параметров нужно, чтобы определить эти распределения? Для каждой из пяти смесей нам нужны `mu` и `log_sigma` (чтобы определить распределение) и вероятность выбора этой смеси (`log_pi`) для каждого из 32 измерений в `z`. То есть $5 \times 3 \times 32 = 480$ параметров. Еще один дополнительный параметр предназначен для прогнозирования вознаграждения, точнее, логарифмических коэффициентов вознаграждения на следующем временном шаге.

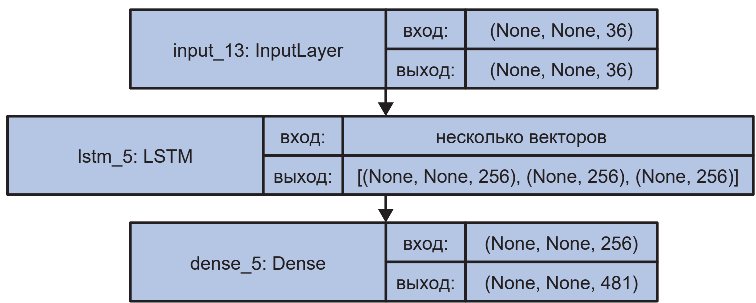



**Рис. 8.13.** Архитектура сети MDN-RNN

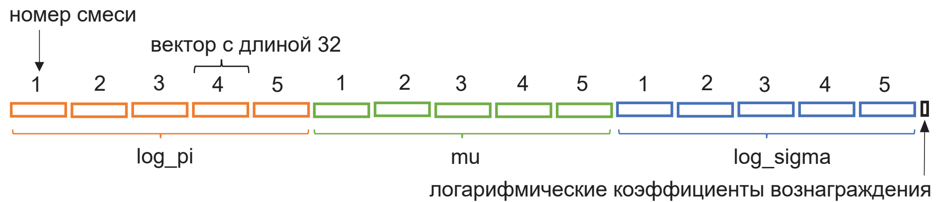



**Рис. 8.14.** Вектор на выходе из MDN-RNN

## Выборка следующего состояния и вознаграждения из MDN-RNN

Мы можем сделать выборку из выходных данных MDN, чтобы сгенерировать прогноз для `z` и вознаграждения на следующем временном шаге, как описано ниже:

1. Разделить 481-мерный выходной вектор на три переменные (`log_pi`, `mu`, `log_sigma`) и значение вознаграждения.
2. Масштабировать вектор `log_pi` так, чтобы его можно было интерпретировать как 32 распределения вероятностей пять индексов смесей.
3. Для каждого из 32 измерений в `z` сделать выборку из распределений, созданных из `log_pi` (то есть выбрать одно из пяти распределений для каждого измерения в `z`).
4. Получить соответствующие значения `mu` и `log_sigma` для этого распределения.
5. Получить выборку значений для каждого измерения в `z` из нормального распределения, параметризованного параметрами `mu` и `log_sigma`, соответствующими этим измерениям.
6. Если значения логарифмических коэффициентов вознаграждений больше 0, прогнозируем вознаграждение `1`; иначе прогнозируем `0`.

## Функция потерь в MDN-RNN

Функция потерь для MDN-RNN — это сумма потерь при реконструкции вектора `z` и вознаграждения.

В листинге 8.3 показана выдержка из файла rnn/arch.py, описывающего архитектуру MDN-RNN, с реализацией пользовательской функции потерь.

**Листинг 8.3.** Выдержка из файла rnn/arch.py

```
def get_responses(self, y_true):

    z_true = y_true[:,:,:Z_DIM]
    rew_true = y_true[:,:,-1]
```

```
    return z_true, rew_true

def get_mixture_coef(self, z_pred):

    log_pi, mu, log_sigma = tf.split(z_pred, 3, 1)
    log_pi = log_pi – K.log(K.sum(K.exp(log_pi), axis = 1, keepdims = True))
    return log_pi, mu, log_sigma

def tf_lognormal(self, z_true, mu, log_sigma):

    logSqrtTwoPI = np.log(np.sqrt(2.0 * np.pi))
    return -0.5 * ((z_true – mu) / K.exp(log_sigma)) ** 2 – log_sigma –
                                                        logSqrtTwoPI

def rnn_z_loss(y_true, y_pred):

    z_true, rew_true = self.get_responses(y_true)

    d = normal distribution_MIXTURES * Z_DIM
    z_pred = y_pred[:,:,:(3*d)]
    z_pred = K.reshape(z_pred, [-1, normal distribution_MIXTURES * 3])

    log_pi, mu, log_sigma = self.get_mixture_coef(z_pred) ❶

    flat_z_true = K.reshape(z_true,[-1, 1])

    z_loss = log_pi + self.tf_lognormal(flat_z_true, mu, log_sigma) ❷
    z_loss = -K.log(K.sum(K.exp(z_loss), 1, keepdims=True))

    z_loss = K.mean(z_loss)

    return z_loss

def rnn_rew_loss(y_true, y_pred):

    z_true, rew_true = self.get_responses(y_true) #, done_true

    d = normal distribution_MIXTURES * Z_DIM
    reward_pred = y_pred[:,:,-1]

    rew_loss = K.binary_crossentropy(rew_true, reward_pred, from_logits = True) ❸

    rew_loss = K.mean(rew_loss)

    return rew_loss

def rnn_loss(y_true, y_pred):

    z_loss = rnn_z_loss(y_true, y_pred)
    rew_loss = rnn_rew_loss(y_true, y_pred)
```

```
    return Z_FACTOR * z_loss + REWARD_FACTOR * rew_loss ❹

opti = Adam(lr=LEARNING_RATE)
model.compile(loss=rnn_loss, optimizer=opti, metrics = [rnn_z_loss,
                                                        rnn_rew_loss])
```

❶ Выходной 481-мерный вектор разбивается на три переменные (`log_pi`, `mu`, `log_sigma`) и значение вознаграждения.

❷ Это расчет потерь при реконструкции вектора `z`: отрицательная логарифмическая вероятность наблюдения истинного `z` в смеси распределений, параметризованной результатом сети MDN-RNN. Для нас желательно, чтобы это значение было как можно больше, что эквивалентно минимизации отрицательного логарифма вероятности.

❸ Для вычисления потерь вознаграждения используется простая бинарная перекрестная энтропия между истинным вознаграждением и прогнозируемыми логарифмическими коэффициентами из сети.

❹ Общая потеря вычисляется как сумма потерь при реконструкции `z` и вознаграждения — весовым параметрам `Z_FACTOR` и `REWARD_FACTOR` здесь присвоено значение `1`, но вообще их можно скорректировать, чтобы придать больше веса потерям при восстановлении или потерям вознаграждения.

Отметим, что для обучения MDN-RNN не потребовалось выбирать конкретные векторы `z` из выходных данных MDN: потери вычисляются непосредственно из 481-мерного выходного вектора.

## Обучение контроллера

Последний этап — обучение контроллера (сети, которая выбирает действие) с использованием эволюционного алгоритма CMA-ES (Covariance Matrix Adaptation Evolution Strategy — эволюционная стратегия с адаптацией матрицы ковариаций). Запустим обучение контроллера, выполнив следующую команду в терминале (вводом в одну строку):

```
xvfb-run -a -s «-screen 0 1400x900x24» python 05_train_controller.py car_racing
  -n 16 -t 2 -e 4 --max_length 1000
```

со следующими параметрами:

`n`

Количество рабочих процессов, которые будут параллельно тестировать решения (их должно быть не больше, чем количество ядер в процессоре на вашем компьютере).

`t`

Количество решений, которые каждый рабочий процесс получит для тестирования в каждом поколении.

`e`

Количество эпизодов, на которых будет проверяться каждое решение для вычисления среднего вознаграждения.

`max_length`

Максимальное количество временных шагов в каждом эпизоде.

`eval_steps`

Количество поколений между оценками текущего наилучшего набора параметров.

Команда, приведенная выше, использует виртуальный буфер (`xvfb`) для отображения кадров, с тем чтобы код мог выполняться на компьютере с ОС Linux без физического экрана. Размер популяции определяется как `pop_size = n * t`.

## Архитектура контроллера

Контроллер имеет очень простую архитектуру. Это полносвязанная нейронная сеть без скрытого слоя; она соединяет входной вектор непосредственно с вектором действия.

Входной вектор — это конкатенация текущего вектора `z` (с длиной 32) и текущего скрытого состояния LSTM (с длиной 256), которая является вектором с длиной 288. Поскольку каждый входной узел подключается непосредственно к трем выходным узлам, общее количество весов равно $288 \times 3 = 864$

плюс три веса смещения, то есть всего 867 весов. Как же обучить эту сеть? Эта задача не относится к категории обучения с учителем — мы не пытаемся *предсказать* правильное действие. У нас нет обучающего набора с *правильными* действиями, так как мы не знаем, какое действие является оптимальным в данном состоянии окружения. Это — отличительная черта задач обучения с подкреплением. Нам нужно, чтобы агент сам нашел оптимальные значения весов, экспериментируя, обновляя веса и опираясь на обратную связь. Эволюционные стратегии пользуются большой популярностью в задачах обучения с подкреплением благодаря их простоте, эффективности и масштабируемости. Мы будем использовать конкретную стратегию, известную как CMA-ES.

## CMA-ES

Эволюционные стратегии обычно придерживаются следующего процесса:

1. Создается *популяция* агентов, и каждый инициализируется случайными значениями параметров.
2. В цикле:
   а) каждый агент действует в окружении и возвращает среднее вознаграждение за несколько эпизодов;
   б) производится селекция агентов с лучшими результатами для создания новых членов популяции;
   в) в параметры новых членов добавляются случайные искажения;
   г) пул популяции агентов обновляется: в него добавляются новые члены и удаляются старые, показавшие худшие результаты.

Все это напоминает процесс эволюции в животном мире. Именно поэтому эти стратегии и называют *эволюционными*. Под «селекцией» здесь подразумевается простое объединение существующих агентов с лучшими результатами, чтобы повысить вероятность получения высоких результатов в следующем поколении. Как и во всех решениях обучения с подкреплением, необходимо найти баланс между жадным поиском локально-оптимальных решений и исследованием неизвестных областей в пространстве параметров, где могут скрываться лучшие решения. Вот почему важно добавить элемент случайности в популяцию, чтобы не сужать поле поиска.

CMA-ES — это всего лишь одна из множества эволюционных стратегий. Она основана на использовании нормального распределения и выбирает параметры для новых агентов из него. В каждом поколении алгоритм стратегии обновляет среднее значение распределения, чтобы максимизировать вероятность выбора агентов с высокими показателями из предыдущего временного шага. Одновременно он обновляет ковариационную матрицу распределения, чтобы максимизировать вероятность выбора агентов с высокой оценкой, опираясь на предыдущее среднее значение. Его можно рассматривать как разновидность градиентного спуска, обладающую дополнительным преимуществом — в ней не используются производные, а это означает, что не нужно производить дорогостоящие вычисления для оценки градиентов. На рис. 8.15 представлено одно поколение алгоритма. Здесь мы пытаемся найти минимум сильно нелинейной двумерной функции — в красной и черной областях функция имеет большие значения, чем в желтой и белой.

Вот эти шаги:

1. Сначала генерируется случайное двумерное нормальное распределение, из которого выбирается популяция кандидатов, изображенных как синие точки.

2. Затем для каждого кандидата вычисляется значение функции и выбираются лучшие 25 %, изображенные как фиолетовые точки, — назовем этот набор точек `P`.

3. По точкам из набора `P` вычисляется среднее значение для нового нормального распределения. Этот шаг можно рассматривать как стадию селекции, в которой выбираются лучшие кандидаты для вычисления среднего нового распределения. По точкам из набора `P` вычисляется ковариационная матрица нового нормального распределения, но с использованием прежнего среднего значения, а не текущего, полученного по набору `P`. Чем больше разница между прежним средним и чем больше точек в наборе `P`, тем больше дисперсия следующего нормального распределения. Это создает эффект импульса в поиске оптимальных параметров.

4. Затем из нового нормального распределения с новыми средним значением и ковариационной матрицей выбирается новая популяция кандидатов.

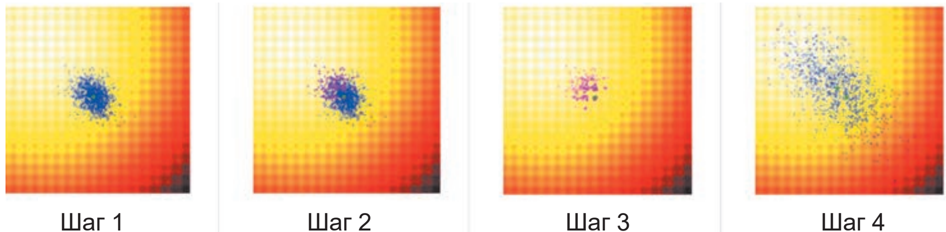



**Рис. 8.15.** Один шаг обновления в алгоритме CMA-ES[1]

На рис. 8.16 показано несколько поколений процесса. Посмотрите, как ковариация расширяется, когда среднее значение движется большими шагами к минимуму, и сужается, когда среднее значение достигает истинного минимума.

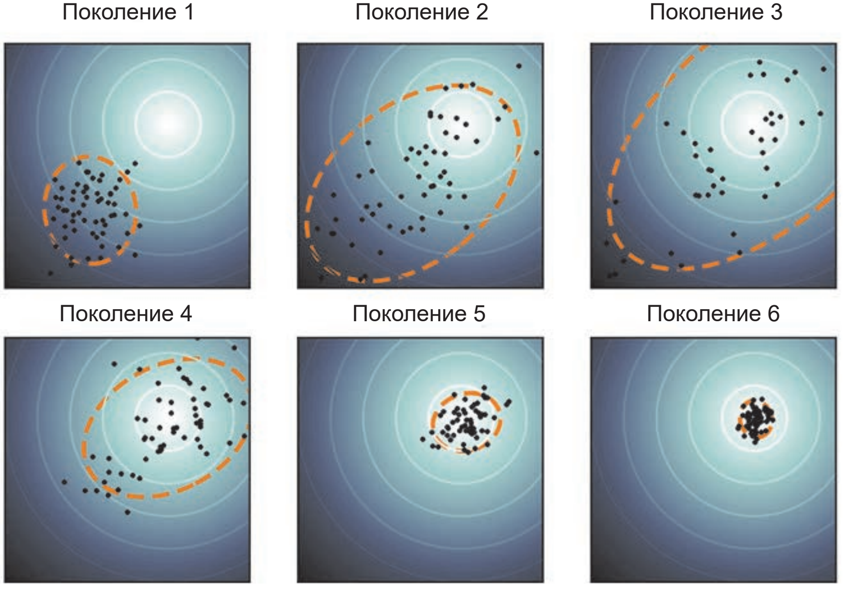



**Рис. 8.16.** CMA-ES[2]

[1] Приводится с разрешения Дэвида Ха (David Ha), 2017, http://bit.ly/2XufRwq.

[2] Источник: https://en.wikipedia.org/wiki/CMA-ES.

В задаче с гоночным автомобилем вместо четко определенной функции, которую можно было бы максимизировать, у нас имеется окружение с 867 параметрами, определяющими, насколько хорошо действует агент. Первоначально некоторые наборы параметров по случайности будут давать высокие оценки, и алгоритм будет постепенно смещать нормальное распределение в направлении параметров с самыми высокими оценками в окружении.

## Параллельное выполнение алгоритма CMA-ES

Одним из больших преимуществ алгоритма CMA-ES является простота его распараллеливания с помощью библиотеки Python, созданной Дэвидом Ха (David Ha), под названием `es.py`. Наиболее трудоемкой частью алгоритма является вычисление оценки для заданного набора параметров, поскольку для этого необходимо смоделировать поведение агента с этими параметрами в окружении. Этот процесс можно распараллелить, так как между отдельными экземплярами агента нет никаких зависимостей. Для этого используем архитектуру ведущий/ведомый, где есть главный (ведущий) процесс, который отправляет наборы параметров для параллельного тестирования множеству подчиненных (ведомых) процессов. Ведомые процессы возвращают результаты ведущему, который накапливает их и затем передает общий результат в объект CMA-ES. Этот объект обновляет среднее значение и ковариационную матрицу нормального распределения, как показано на рис. 8.15, и передает ведущему процессу новую популяцию для тестирования. Затем цикл повторяется. Диаграмма на рис. 8.17 поясняет происходящее.

❶ Ведущий процесс запрашивает у объекта CMA-ES (`es`) набор параметров для опробования.

❷ Ведущий процесс распределяет параметры между ведомыми процессами. Здесь каждый из четырех ведомых процессов получает два набора параметров.

❸ Ведомые процессы запускают рабочие потоки, которые в цикле перебирают набор параметров и для каждого выполняют несколько эпизодов. Каждый набор параметров опробуется на трех эпизодах.

❹ Для вознаграждений во всех эпизодах вычисляется среднее, которое служит оценкой для каждого набора параметров.

❺ Ведомый процесс возвращает список оценок ведущему.

❻ Ведущий процесс группирует оценки и посылает полученный список объекту `es`.

❼ Объект `es` вычисляет по списку оценок новое нормальное распределение (см. вновь рис. 8.15).

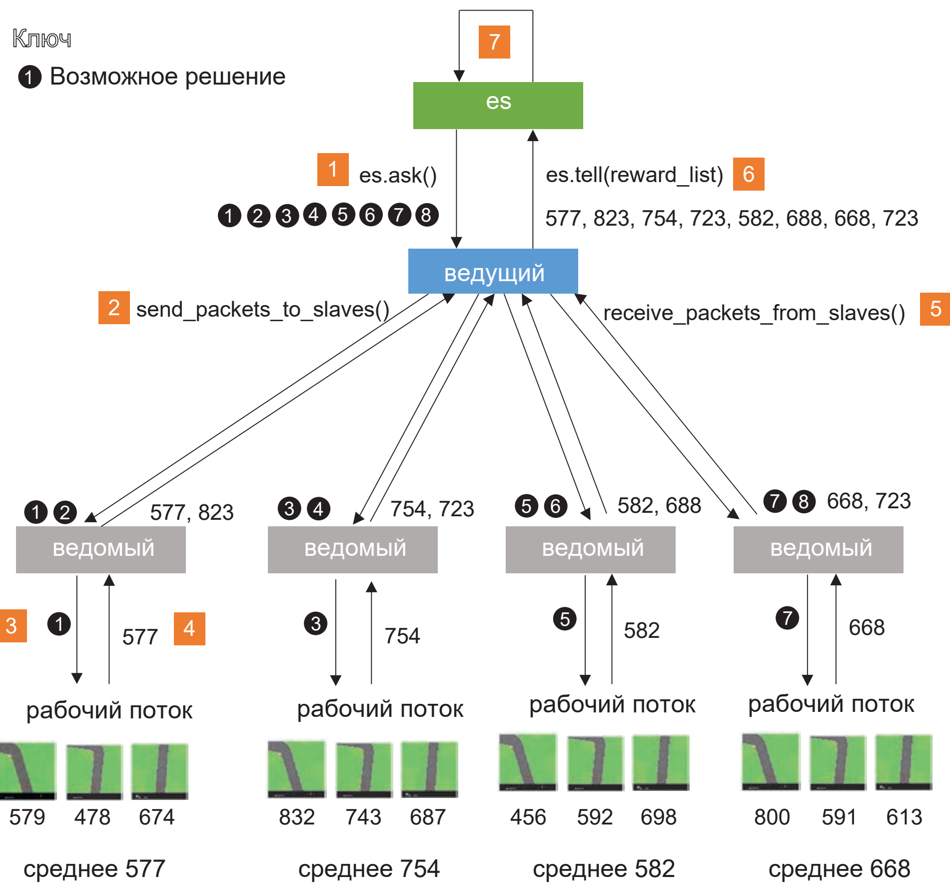



**Рис. 8.17.** Параллельное выполнение алгоритма CMA-ES — здесь имеется 8 популяций и 4 ведомых процесса (поэтому каждый ведомый процесс выполняет t = 2 испытаний)

## Вывод контроллера в процессе обучения

На рис. 8.18 показан вывод контроллера в процессе обучения. Веса обученной сети сохраняются в файле через каждые `eval_steps` поколений.

Каждая строка в выводе представляет одно поколение, сгенерированное в ходе обучения. Для каждого поколения выводятся следующие статистики:

1. Имя окружения (например, `car_racing`).
2. Номер поколения (например, `16`).
3. Текущее прошедшее время в секундах (например, `2395`).
4. Среднее вознаграждение, полученное поколением (например, `136.44`).
5. Минимальное вознаграждение, полученное поколением (например, `33.28`).
6. Максимальное вознаграждение, полученное поколением (например, `246.12`).
7. Стандартное отклонение вознаграждения (например, `62.78`).

```
('car_racing', (1, 151, -77.57, -93.47, -57.68, 13.43, 0.49663, 1000.0, 1000))
('car_racing', (2, 298, -64.16, -82.57, -26.64, 14.17, 0.49346, 1000.0, 1000))
('car_racing', (3, 449, -65.13, -76.36, -53.9, 7.41, 0.49046, 1000.0, 1000))
('car_racing', (4, 595, -63.1, -92.23, -28.59, 16.03, 0.48763, 1000.0, 1000))
('car_racing', (5, 743, -52.26, -72.98, 1.18, 20.67, 0.48492, 1000.0, 1000))
('car_racing', (6, 892, -47.26, -74.36, -0.07, 21.21, 0.48233, 1000.0, 1000))
('car_racing', (7, 1039, -36.81, -57.4, -19.15, 13.37, 0.47984, 1000.0, 1000))
('car_racing', (8, 1189, -35.75, -76.86, 13.82, 26.54, 0.47744, 1000.0, 1000))
('car_racing', (9, 1338, -33.7, -79.83, 28.03, 30.95, 0.47511, 1000.0, 1000))
('car_racing', (10, 1485, -17.43, -63.65, 69.14, 36.05, 0.47284, 1000.0, 1000))
('car_racing', (11, 1632, -10.99, -63.11, 106.43, 43.87, 0.47063, 1000.0, 1000))
('car_racing', (12, 1781, 7.1, -70.57, 65.42, 38.67, 0.46848, 1000.0, 1000))
('car_racing', (13, 1933, 38.88, -61.85, 197.08, 76.47, 0.46638, 1000.0, 1000))
('car_racing', (14, 2085, 72.03, -31.57, 147.58, 43.74, 0.46433, 1000.0, 1000))
('car_racing', (15, 2238, 95.18, -15.08, 260.42, 75.69, 0.46234, 1000.0, 1000))
('car_racing', (16, 2395, 136.44, 33.28, 246.12, 62.78, 0.4604, 1000.0, 1000))
('car_racing', (17, 2549, 133.55, -1.81, 248.34, 62.98, 0.45854, 1000.0, 1000))
('car_racing', (18, 2702, 172.56, 64.58, 237.96, 55.28, 0.45672, 1000.0, 1000))
('car_racing', (19, 2847, 171.07, 51.86, 308.79, 80.94, 0.45497, 1000.0, 1000))
```

**Рис. 8.18.** Вывод контроллера в процессе обучения

8. Текущий коэффициент стандартного отклонения процесса эволюционной стратегии (инициализируется значением `0.5` и затем уменьшается на каждом временном шаге; например `0.4604`).

9. Минимальное число временных шагов до завершения (например, `1000.0`).

10. Максимальное число временных шагов до завершения (например, `1000.0`).

После `eval_steps` временных шагов каждый ведомый процесс определяет набор параметров с лучшей оценкой и возвращает среднее вознаграждение по нескольким эпизодам. Эти значения затем снова усредняются и возвращаются как общая оценка набора параметров. Примерно через 200 временных шагов процесс обучения в задаче управления гоночным автомобилем достиг средней величины вознаграждения 840.

## Обучение в мнимом окружении

До настоящего времени обучение контроллера проводилось с использованием метода `step` окружения `CarRacing` из OpenAI Gym, который реализует переход модели из одного состояния в другое. Эта функция вычисляет следующее состояние и вознаграждение, исходя из текущего состояния окружения и выбранного действия. Отметим, что функционально метод `step` очень похож на сеть MDN-RNN в нашей модели. MDN-RNN выводит прогноз `z` и вознаграждения, исходя из текущего `z` и выбранного действия.

Фактически MDN-RNN можно рассматривать как самостоятельное окружение, работающее в пространстве `z`, а не в пространстве исходных изображений. Невероятно, но это значит, что мы можем заменить реальное окружение копией сети MDN-RNN и обучить контроллер, используя исключительно сеть MDN-RNN, которая *представляет*, как должно вести себя окружение.

Иными словами, сеть MDN-RNN достаточно хорошо изучила физику реального окружения на исходном наборе данных с информацией о случайных перемещениях и теперь ее можно использовать в качестве заменителя реального окружения для обучения контроллера. Это означает, что агент может научиться решать новые задачи, *стремясь* максимизировать вознаграждение в мнимом окружении, без необходимости проверять стратегии

в реальном мире. После этого он сможет хорошо выполнить задачу с первого раза, не выполняя ее в реальности до этого ни разу. Это объясняет, почему статья «World Models» так важна и почему генеративное моделирование почти наверняка станет ключевым компонентом искусственного интеллекта в будущем.

Ниже приводится сравнение архитектур для обучения в реальном и мнимом окружениях: архитектура с реальным окружением показана на рис. 8.19, а с мнимым — на рис. 8.20.

Обратите внимание, что в архитектуре с мнимым окружением обучение контроллера происходит полностью в пространстве z и нет необходимости декодировать векторы z обратно в изображения трека. Конечно, мы

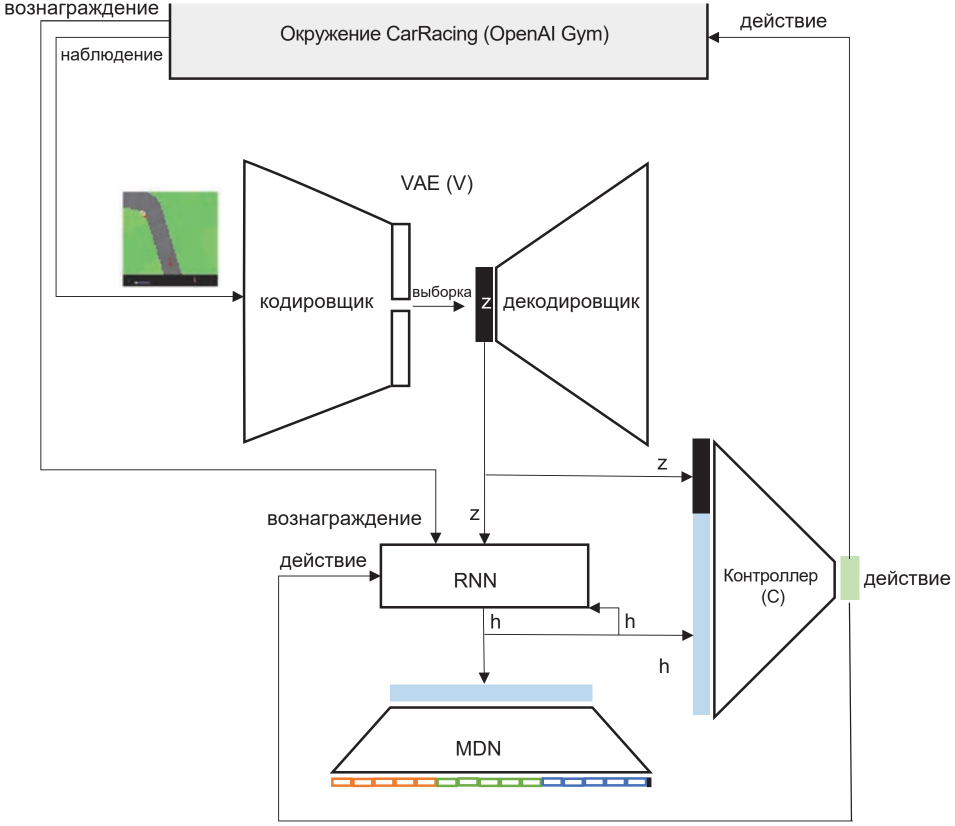



**Рис. 8.19.** Обучение контроллера в окружении из OpenAI Gym

могли бы сделать это, чтобы визуально проверить работу агента, но для обучения этого не требуется.

## Обучение контроллера в мнимом окружении

Чтобы обучить контроллер в мнимом окружении, выполним следующую команду в терминале (вводом в одну строку):

```
xvfb-run -a -s «-screen 0 1400x900x24» python 05_train_controller.py car_racing
  -n 16 -t 2 -e 4 --max_length 1000 --dream_mode 1
```

Эта же команда использовалась для обучения контроллера в реальном окружении с дополнительным параметром `--dream_mode 1`. Вывод процесса обучения показан на рис. 8.21.

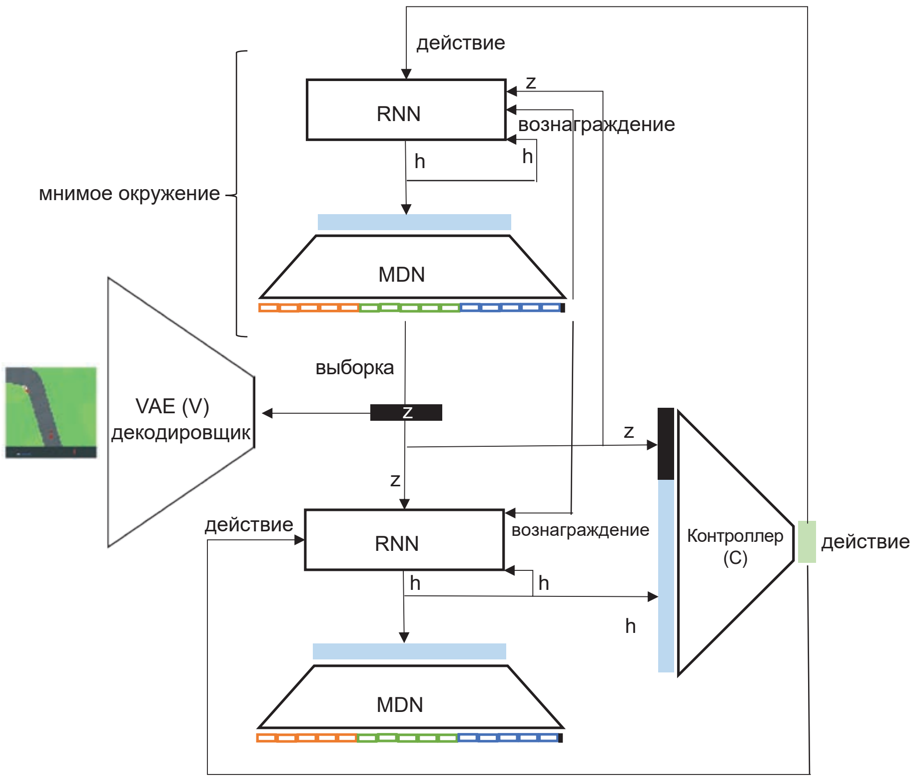



**Рис. 8.20.** Обучение контроллера в мнимом окружении сети MDN-RNN

При обучении в мнимом окружении оценками поколений служат средние суммы мнимых вознаграждений (то есть 0 или 1 на каждом временном шаге). Вычисление оценки через каждые 10 поколений все еще производится в реальном окружении и поэтому определяется на основе суммы наград из окружения OpenAI Gym, что дает возможность сравнить методы обучения.

Спустя всего 10 поколений в мнимом окружении средняя оценка агента составила 586,6 по шкале реального окружения. Агент способен вести автомобиль точно по треку и успешно справляется с большинством поворотов, кроме особенно крутых. Это удивительное достижение — напомним, что когда контроллер оценивался после обучения 10 поколений в реальном окружении, он *никогда* не пытался ехать по треку быстро. Он просто выбирал случайные действия, управляя автомобилем в реальном окружении (для обучения VAE и MDN-RNN), а затем, уже в своем мнимом окружении, обучил контроллер.

```
('car_racing', (1, 69, 2.24, 1.0, 10.25, 2.14, 0.09895, 1000.0, 1000))
('car_racing', (2, 133, 5.52, 1.0, 21.25, 4.28, 0.098, 1000.0, 1000))
('car_racing', (3, 196, 16.81, 1.0, 71.75, 16.87, 0.09713, 1000.0, 1000))
('car_racing', (4, 261, 17.18, 1.5, 71.5, 16.79, 0.0963, 1000.0, 1000))
('car_racing', (5, 324, 29.7, 1.25, 118.0, 28.6, 0.09553, 1000.0, 1000))
('car_racing', (6, 388, 46.91, 5.75, 136.75, 35.33, 0.09481, 1000.0, 1000))
('car_racing', (7, 453, 65.85, 5.5, 152.0, 40.97, 0.09412, 1000.0, 1000))
('car_racing', (8, 518, 105.47, 15.75, 203.5, 41.4, 0.09348, 1000.0, 1000))
('car_racing', (9, 584, 123.28, 40.75, 206.5, 36.28, 0.09287, 1000.0, 1000))
('car_racing', (10, 650, 139.37, 70.5, 218.5, 38.2, 0.09228, 1000.0, 1000))
[349.541  489.1644 644.3677 708.0521 495.2128 647.6338 530.0317 515.0535
 661.6697 527.1665 720.0582 665.7434 646.2141 637.3213 572.7351 629.6684
 676.5256 529.0831 626.0878 575.0685 626.9549 664.0143 632.5556 599.2186
 326.8668 599.7362 694.7311 652.5841 603.1318 659.2366 447.6374 418.6205]
[586.61520625]
('improvement', 10, 584.37520625, 'curr', 586.61520625, 'prev', 2.24, 'best', 586.61520625)
('car_racing', (11, 975, 168.35, 99.25, 213.5, 24.7, 0.0917, 1000.0, 1000))
('car_racing', (12, 1039, 163.49, 70.25, 253.5, 38.67, 0.09116, 1000.0, 1000))
('car_racing', (13, 1104, 184.42, 80.75, 268.0, 41.62, 0.09065, 1000.0, 1000))
('car_racing', (14, 1168, 200.78, 122.75, 265.5, 29.68, 0.09016, 1000.0, 1000))
('car_racing', (15, 1232, 205.96, 80.75, 261.5, 44.21, 0.08969, 1000.0, 1000))
('car_racing', (16, 1297, 201.39, 96.0, 260.25, 44.5, 0.08923, 1000.0, 1000))
('car_racing', (17, 1360, 221.05, 101.25, 293.75, 35.19, 0.08878, 1000.0, 1000))
('car_racing', (18, 1426, 236.72, 159.25, 281.25, 28.93, 0.08835, 1000.0, 1000))
('car_racing', (19, 1492, 226.21, 75.5, 283.5, 42.45, 0.08793, 1000.0, 1000))
('car_racing', (20, 1558, 233.88, 119.75, 300.75, 42.47, 0.08752, 1000.0, 1000))
[387.4787 111.7849 417.2107 533.8831 448.2589 536.753  204.1228 188.0443
 333.7994 335.6777 486.4987 306.7614 252.5269 247.3025 417.719  212.4605
 511.8903 353.9119 398.2054 516.7442 374.3491 624.1784 413.6094 372.7289
 262.7903 449.3083 322.7305 324.8756 301.511  299.5016 388.2149 302.5429]
[363.667975]
('improvement', 20, -222.94723125000002, 'curr', 363.667975, 'prev', 586.61520625, 'best', 586.61520625)
```

**Рис. 8.21.** Вывод процесса обучения в мнимом окружении

Для сравнения, после обучения 10 поколений в реальном окружении агент был едва способен двинуться вперед с линии старта. Более того, обучение каждого поколения в мнимом окружении происходит примерно в три-четыре раза быстрее, потому что прогнозирование `z` и вознаграждения в MDN-RNN происходит быстрее, чем вычисление `z` и вознаграждения в окружении OpenAI Gym.

## Недостатки обучения в мнимом окружении

Одним из недостатков обучения агентов полностью в мнимом окружении MDN-RNN является переобучение. Это происходит, когда агент обнаруживает беспроигрышную стратегию в мнимом окружении, но не обобщает ее на реальное окружение из-за того, что MDN-RNN не смогла изучить поведение истинного окружения в определенных условиях. Это можно заметить на рис. 8.21: после 20 поколений, даже при том, что оценка в мнимом окружении продолжает расти, в реальном окружении агент получил оценку всего лишь 363,7, что меньше, чем после обучения 10 поколений.

Авторы оригинальной статьи «World Models» подчеркивают эту проблему и показывают, как включение параметра `temperature` для управления неопределенностью в модели может помочь решить ее. Увеличение этого параметра влечет увеличение дисперсии при выборе `z` из MDN-RNN и уменьшение стабильности прогона при обучении в мнимом окружении. Контроллер получает высокие вознаграждения за более безопасные стратегии, которые сталкиваются с хорошо известными состояниями и, следовательно, лучше обобщают реальное окружение. Однако параметр `temperature` не может увеличиваться бесконечно, иначе окружение получится настолько изменчивым, что контроллер не сможет прийти к какой-то определенной стратегии из-за нехватки согласованности в том, как мнимое окружение меняется с течением времени.

В статье описан пример успешного применения этого приема в другом окружении: `DoomTakeCover`, основанном на компьютерной игре *Doom*. На рис. 8.22 показано, как изменение параметра `temperature` влияет на виртуальную (в мнимом окружении) и на фактическую (в реальном окружении) оценки.

| Temperature $\tau$ | Virtual Score | Actual Score |
|---|---|---|
| 0.10 | $2086 \pm 140$ | $193 \pm 58$ |
| 0.50 | $2060 \pm 277$ | $196 \pm 50$ |
| 1.00 | $1145 \pm 690$ | $868 \pm 511$ |
| 1.15 | $918 \pm 546$ | $1092 \pm 556$ |
| 1.30 | $732 \pm 269$ | $753 \pm 139$ |
| Random Policy | N/A | $210 \pm 108$ |
| Gym Leader | N/A | $820 \pm 58$ |

**Рис. 8.22.** Использование параметра temperature для управления изменчивостью мнимого окружения[1]

## Итоги

В этой главе мы узнали, как использовать генеративную модель (VAE) в условиях обучения с подкреплением, чтобы позволить агенту найти эффективную стратегию путем проверки правил в пределах собственного сгенерированного мнимого, а не реального окружения. Вариационный автокодировщик (VAE) учится изучать скрытое представление окружения, которое затем передается на вход рекуррентной нейронной сети, прогнозирующей будущие траектории в скрытом пространстве.

Удивительно, но агент может затем использовать эту генеративную модель в качестве мнимого окружения для итеративной проверки правил, используя эволюционную методологию, которая хорошо обобщает реальное окружение.

[1] Источник: Ха (Ha) и Шмидхубер (Schmidhuber), 2018.

# Глава 9

# Будущее генеративного моделирования

Я начал писать эту книгу в мае 2018 года, вскоре после публикации статьи «World Models» (глава 8), стремясь, чтобы именно эта статья заняла центральное место в последней, основной главе книги, потому что она описывает самый первый практический пример того, как генеративные модели могут способствовать развитию более глубокой формы обучения, протекающего в модели окружения внутри агента. Но и сейчас я нахожу этот пример совершенно поразительным. Это взгляд в будущее, когда агенты будут учиться не только за счет максимизации вознаграждения в окружении по нашему выбору, но и за счет создания собственного представления окружения и, как следствие, получат возможность создавать собственные функции вознаграждения для оптимизации. В этой главе мы разберем эту идею и посмотрим, куда она нас приведет.

Сначала мы встанем на передний край генеративного моделирования, где находятся самые радикальные и инновационные идеи. Замечу, что уже после начала работы над книгой в генеративно-состязательных сетях и методологиях, основанных на внимании, были достигнуты значительные успехи, приведшие к тому, что теперь мы можем генерировать изображения, текст и музыку, практически неотличимые от произведений, созданных человеком. Мы начнем с представления этих достижений на примерах, которые

уже исследовали, и пройдемся по самым передовым архитектурам, доступным в настоящее время.

## Пять лет прогресса

История генеративного моделирования коротка, если сравнивать ее с более широко изученным дискриминационным моделированием — изобретение генеративно-состязательных сетей (GAN) в 2014 году можно, пожалуй, рассматривать как искру, разжигающую пламя. На рис. 9.1 показана сводная информация о ключевых событиях в генеративном моделировании, многие из которых мы уже исследовали в этой книге.

Это ни в коем случае не исчерпывающий список — существуют десятки разновидностей генеративно-состязательных сетей, новаторских в своих областях (например, генерирующие видео или преобразующие текст в изображение). Здесь я покажу подборку самых последних разработок, которые раздвинули границы генеративного моделирования в целом.

С середины 2018 года произошел целый ряд примечательных событий в области генеративного моделирования на основе последовательностей и изображений. Моделирование последовательностей по большей части основано на изобретении трансформера, механизма внимания, который полностью устраняет необходимость в использовании рекуррентных или сверточных нейронных сетей и в настоящее время поддерживается большинством современных последовательных моделей (BERT, GPT-2 и MuseNet). Генерирование изображений достигло новых высот благодаря разработке новых технологий на основе генеративно-состязательных сетей (ProGAN, SAGAN, BigGAN и StyleGAN).

Детальное описание этих событий и их последствий может легко заполнить целую книгу. Поэтому в этой главе мы рассмотрим их лишь настолько, чтобы понять базовые идеи, лежащие в основе современного генеративного моделирования. Затем, вооружившись этим знанием, мы попробуем понять, в каких направлениях будет развиваться эта область в ближайшем будущем, и представить, чего удастся достигнуть в ближайшие годы.

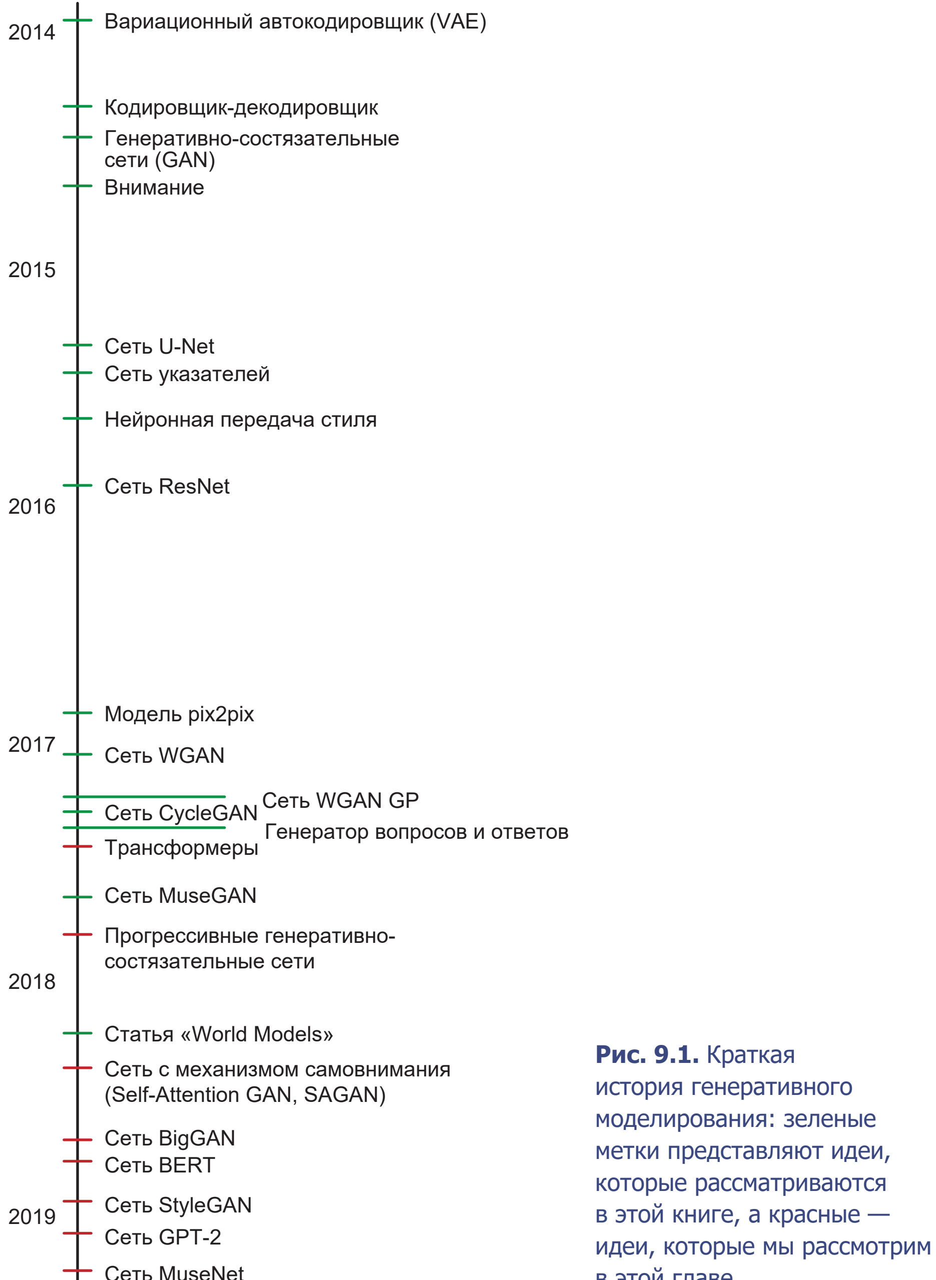



**Рис. 9.1.** Краткая история генеративного моделирования: зеленые метки представляют идеи, которые рассматриваются в этой книге, а красные — идеи, которые мы рассмотрим в этой главе

## Трансформер

Впервые механизм трансформера описан в статье «Attention is All You Need»[1], где показано, как создавать мощные нейронные сети для последовательного моделирования, не требующие сложных рекуррентных или сверточных архитектур и полагающиеся только на механизмы внимания. К настоящему времени представлено несколько впечатляющих практических примеров генеративного моделирования вроде BERT и GPT-2, созданных в Google для решения языковых задач, и MuseNet для генерирования музыки. На рис. 9.2 показана обобщенная архитектура трансформера.

Авторы применили модель трансформера для перевода текстов с английского на немецкий и с английского на французский. Трансформер имеет типичную для моделей перевода архитектуру кодировщик-декодировщик (описанную в главе 6), но, в отличие от них, вместо рекуррентного слоя, подобного LSTM внутри кодировщика и декодировщика, трансформер использует последовательности слоев внимания.

Слева на рис. 9.2 показан набор из N = 6 слоев внимания, кодирующих входное предложение $\mathbf{x} = (x_1, ..., x_n)$ в последовательное представление. Затем декодировщик, справа на рис. 9.2, генерирует выходные слова по одному, используя результат этого кодирования и предыдущие слова как дополнительные входные данные. Чтобы понять, как действует этот механизм, рассмотрим по шагам процесс обработки входной последовательности.

### Позиционное кодирование

Сначала слова передаются в слой `Embedding`, чтобы преобразовать каждое из них в вектор с длиной $d_{model} = 512$. Так как в этой архитектуре отсутствует рекуррентный слой, необходимо закодировать положение каждого слова в предложении посредством функции позиционного кодирования, преобразующей позицию *pos* слова в предложении в вектор с длиной $d_{model}$:

$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right),$$

$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{(2i+1)/d_{model}}}\right).$$

[1] Ашиш Васвани (Ashish Vaswani) и др., «Attention Is All You Need», 12 июня 2017, https://arxiv.org/abs/1706.03762.

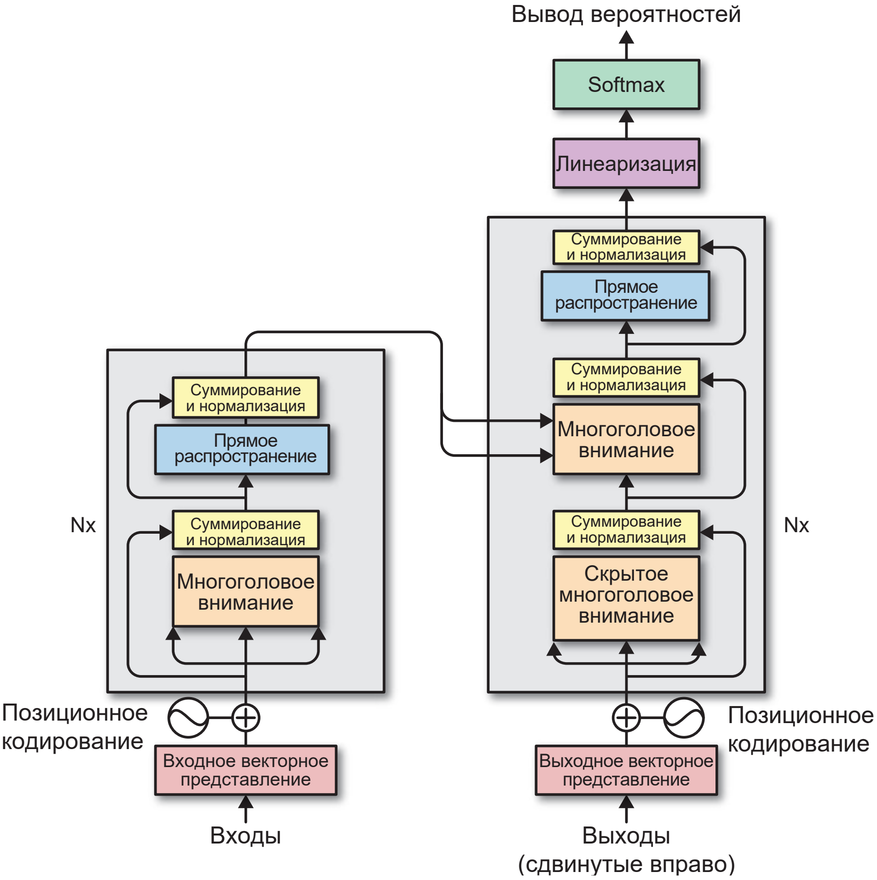



**Рис. 9.2.** Архитектура модели трансформера[1]

Для малых значений *i* эта функция имеет короткий период, поэтому ее значение быстро изменяется вдоль оси, соответствующей позиции. С увеличением значения *i* увеличивается период функции, поэтому для близких друг к другу слов генерируются примерно одинаковые значения. Таким образом, каждая позиция получает свой уникальный код, а поскольку функция может применяться к любым значениям *pos*, ее можно использовать для кодирования любых позиций, независимо от длины входной последовательности. Чтобы создать входные данные для первого слоя кодировщика, матрица

[1] Источник: Васвани и др., 2017.

позиционного кодирования складывается с матрицей векторных представлений слов (рис. 9.3). При таком подходе смысл и положение каждого слова в последовательности фиксируются в одном векторе с длиной $d_{model}$.

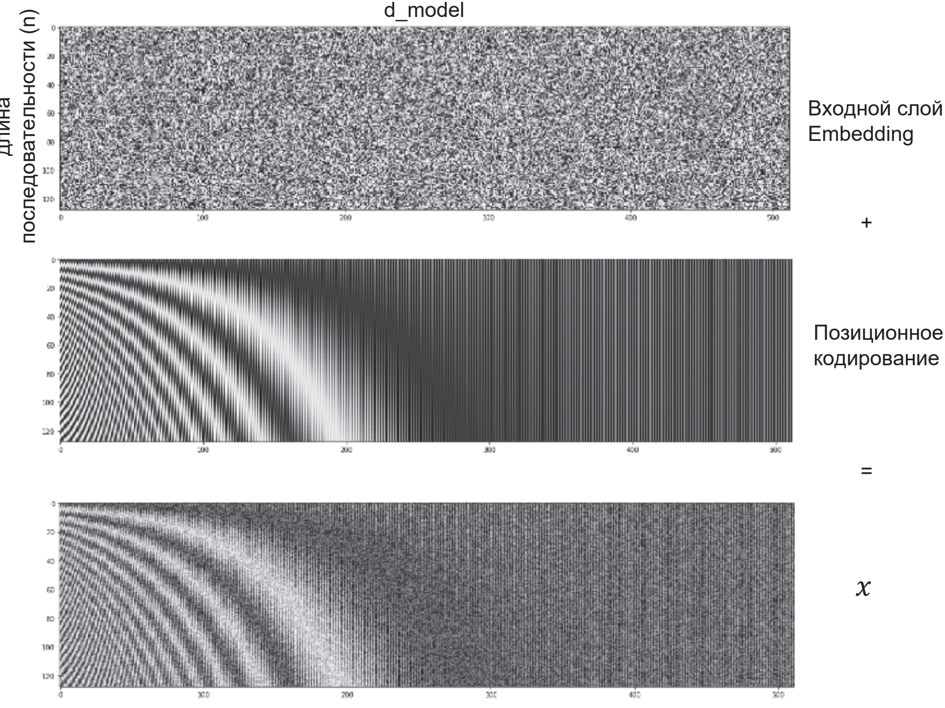



**Рис. 9.3.** Входная матрица векторных представлений слов складывается с матрицей позиционного кодирования, чтобы получить входные данные для первого слоя кодировщика

## Многоголовое внимание

Затем тензор передается в первый из шести слоев кодировщика. Каждый слой кодировщика состоит из нескольких вложенных слоев, начиная со слоя *многоголового внимания (multihead attention layer)*.

Кодировщик и декодировщик имеют одну и ту же архитектуру многоголового внимания с небольшими отличиями. В общем виде архитектура показана на рис. 9.4.

Слой многоголового внимания имеет два входа: тензор запроса $x^Q$ и тензор ключей/значений $x^{KV}$. Задача слоя — в том, чтобы узнать, на какие позиции в тензоре с ключами/значениями следует обратить внимание для каждой позиции в запросе. Ни одна из весовых матриц слоя не зависит от длины тензора запроса ($n^Q$) или тензора с ключами/значениями ($n^{KV}$), поэтому слой может обрабатывать последовательности произвольной длины. Кодировщик использует механизм *самовнимания (self-attention)*, то есть запрос и ключи/значения остаются неизменными (в том виде, в каком были получены от предыдущего слоя в кодировщике). Например, на оба входа первого слоя кодировщика подается результат объединения векторных представлений слов с матрицей позиционного кодирования. В декодировщике входной запрос поступает от предыдущего слоя декодировщика, а ключи/значения — от последнего слоя кодировщика. На первом шаге слой создает три матрицы: запроса $Q$, ключей $K$ и значений $V$, умножая вход на три весовые матрицы, $W^Q$, $W^K$ и $W^V$:

$$Q = x^Q W^Q \qquad K = x^{KV} W^K \qquad V = x^{KV} W^V$$

$Q$ и $K$ являются представлениями запроса и ключей/значений соответственно. Мы должны оценить сходство этих представлений для каждой позиции запроса и ключа/значения. Сделать это можно, умножив матрицу $Q$ на $K^T$ и применив масштабирующий коэффициент $\sqrt{d_k}$. Подобный метод называется *вниманием взвешенного скалярного произведения (scaled dot-product attention)*. Масштабирование играет важную роль, гарантируя, что скалярное произведение векторов $Q$ и $K$ не получится слишком большим.

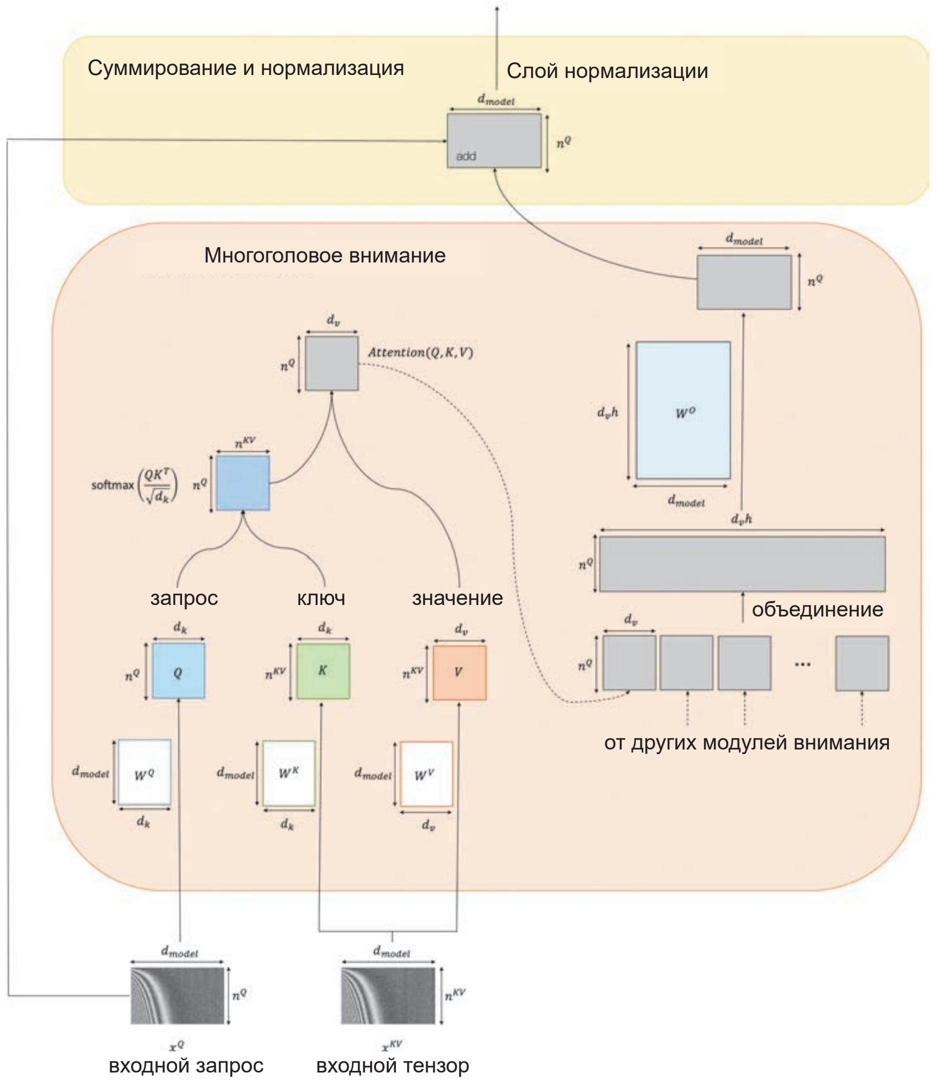



**Рис. 9.4.** Организация модуля многоголового внимания, за которым следует слой суммирования и нормализации

Затем применяется функция *softmax*, чтобы гарантировать, что сумма всех строк будет равна 1. Эта матрица имеет форму $n_Q \times n_{KV}$ и эквивалентна матрице внимания на рис. 7.10. Последний шаг, завершающий действия единственного модуля внимания, — умножение матрицы внимания на матрицу значения $V$. Другими словами, модуль выводит взвешенную сумму представлений значения $V$ для каждой позиции в запросе, где веса определяются матрицей внимания.

Нет, впрочем, никаких причин ограничиваться единственным модулем внимания. В статье авторы используют восемь модулей, которые обучаются параллельно и каждый из которых выводит матрицу $n_Q \times d_v$. Использование нескольких модулей позволяет каждому изучить свой механизм внимания и значения, что обогащает вывод слоя внимания с несколькими модулями.

Выходные матрицы из нескольких модулей внимания объединяются и умножаются на матрицу весов $W^O$. Затем результат складывается поэлементно со входным запросом через пропускающие соединения (skip connections), и к результату применяется слой нормализации (см. рис. 5.7).

Последним элементом кодировщика является слой прямого распространения (полносвязанный), применяемый отдельно к каждой позиции. Веса распределяются между позициями, но не между слоями кодировщика-декодировщика. Кодировщик завершается одним последним пропускающим соединением и слоем нормализации. Обратите внимание, что выход слоя имеет ту же форму, что и входной запрос ($n_Q \times d_{model}$). Это позволяет наложить друг на друга несколько слоев кодировщика, благодаря чему модель получает способность изучать более глубокие признаки.

## Декодировщик

Слои в декодировщике очень похожи на слои в кодировщике — с учетом двух важных отличий:

1. Начальный слой самовнимания — маскированный, вследствие чего информация из последующих временных шагов не используется во время обучения. Маскировка осуществляется установкой соответствующих элементов входа в *softmax* равными $-\infty$.

2. Выход слоя кодировщика также включается в каждый слой декодировщика после начального механизма самовнимания. Здесь входной запрос

поступает из предыдущего слоя декодировщика, а ключи/значения — из кодировщика.

Каждая позиция на выходе из последнего слоя декодировщика передается в последний полносвязанный слой с функцией активации *softmax*, чтобы получить вероятность следующего слова.

## Анализ трансформера

В репозитории GitHub проекта Tensorflow имеется блокнот Colab (http://bit.ly/2HPw4Cw), в котором можно поэкспериментировать с обученной моделью трансформера и посмотреть, как механизмы внимания кодировщика и декодировщика влияют на перевод заданного предложения на немецкий язык. Например, на рис. 9.5 показано, как два модуля внимания в слое декодировщика работают вместе, чтобы дать правильный перевод на немецкий язык

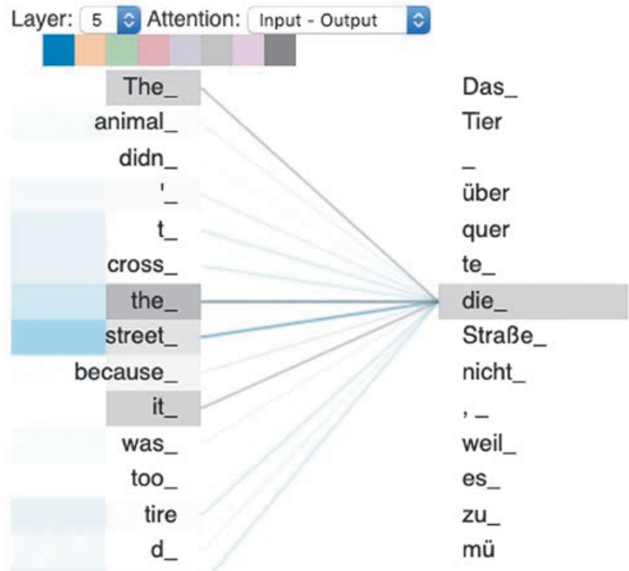



**Рис. 9.5.** Пример того, как один модуль внимания следит за словом «the», а другой — за словом «street», чтобы правильно перевести английский артикль «the» в немецкий «die» — определенный артикль женского рода для слова «Straße»

артикля *the*, когда он используется в контексте *the street*. В немецком языке есть три определенных артикля (*der*, *die*, *das*), которые используются в зависимости от рода существительного, но трансформер знает, что должен выбрать артикль *die*, потому что один модуль внимания следит за словом *street* (в немецком языке имеет женский род), в то время как другой следит за переводимым словом (*the*).

Возможность включать информацию из нескольких мест во входном предложении и в текущем переводе для выбора следующего слова позволяет трансформеру переводить чрезвычайно сложные и длинные предложения. Архитектура трансформера стимулировала появление нескольких последующих моделей, которые используют механизм внимания с несколькими модулями. Далее мы кратко рассмотрим некоторые из них.

## BERT

BERT (Bidirectional Encoder Representations from Transformers — двунаправленный кодировщик представлений на основе трансформера)[1] — модель, разработанная в Google и предсказывающая пропущенные слова в предложении с учетом контекста (до и после пропущенного слова). Это достигается с помощью *модели языка с маской*: во время обучения 15 % слов маскируются случайным образом, а модель должна попытаться воссоздать исходное предложение по маскированному. Важно отметить, что 10 % лексем, выбранных для маскировки, на самом деле заменяются другим словом, а не лексемой `<MASK>`, поэтому модель должна научиться не только заменять лексемы `<MASK>` реальными словами, но и находить во входном предложении слова, не соответствующие контексту, которые могут оказаться словами, заменившими реальные слова.

Модель BERT превосходит свои аналоги типа GloVe, получая представления слов, изменяющиеся в зависимости от контекста. Так, в английском языке слово *water* (вода) может использоваться как глагол (*I need to water the plant* — мне нужно полить рассаду) или существительное (*The ocean is full of water* — океан полон воды). В модели GloVe векторы определяют одно и то же представление для слова *water* независимо от контекста, тогда как BERT

[1] Джейкоб Девлин (Jacob Devlin) и др., «BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding», 11 октября 2018, https://arxiv.org/abs/1810.04805v1.

добавляет информацию об окружении, чтобы получить индивидуальное представление слова в контексте.

Модель BERT позволяет добавлять надстройки — выходные слои для решения конкретной задачи. Например, для решения задач классификации, таких как анализ эмоциональной окраски, можно добавить слой, принимающий вывод трансформера, а для генерирования вопросов и ответов можно организовать маркировку ответов во входной последовательности, использовав сеть указателей в качестве выходного слоя для BERT. Взяв за основу предварительно обученную модель BERT и настраивая дополнительные выходные слои, можно быстро обучить чрезвычайно сложные языковые модели для различных задач моделирования.

## GPT-2

Модель GPT-2, разработанная OpenAI, обучена предсказывать следующее слово в отрывке текста. Если модель BERT была ответом компании Google на появление ранней версии модели OpenAI GPT, то GPT-2 — это прямой ответ на BERT. В отличие от двунаправленной модели BERT, GPT-2 является однонаправленной. Таким образом, для формирования представления текущего слова GPT-2 не использует информацию из слов, следующих за ним, и поэтому предназначена для решения задач генерирования предложений, таких как рассматривавшаяся в главе 6 задача генерирования текста в стиле басен Эзопа. На рис. 9.6 показан пример текста, полученного моделью GPT-2 на основе начального предложения, предложенного системой.

Если вас немного напугала реалистичность полученного текста, то знайте, что вы не одиноки. Из-за опасений, что эта модель может быть использована злоумышленниками, например, для создания поддельных новостей, фальшивых статей, фиктивных учетных записей в социальных сетях или подделки под людей в онлайн-общении в интернете, в OpenAI решили не публиковать набор данных, код и веса модели GPT-2. Вместо этого они официально выпустили только малую (117 миллионов параметров) и среднюю (345 миллионов параметров) версии GPT-2 (http://bit.ly/2WShm7t).[1]

[1] По состоянию на май 2019 года для доверенных партнеров, занимающихся исследованием возможного влияния таких сложных моделей и способов противодействия неправильному использованию, была выпущена версия GPT-2 с полутора миллиардами параметров.

## MuseNet

MuseNet (http://bit.ly/31hT2vl) — модель, также выпущенная в OpenAI и применяющая архитектуру трансформера для генерации музыки. Как и GPT-2, она является однонаправленной и обучена предсказывать следующую ноту, исходя из последовательности предыдущих нот.

В задачах генерирования музыки длина последовательности $N$ увеличивается по мере продвижения вперед, то есть вычислять и хранить матрицу $N \times N$ для каждого модуля внимания становится очень дорого. Но мы не можем просто взять и обрезать входную последовательность, потому что желательно, чтобы модель строила фрагменты, опираясь на долгосрочную структуру, и повторяла мотив и фразы, отстоящие на несколько минут назад, как это часто делают живые композиторы.

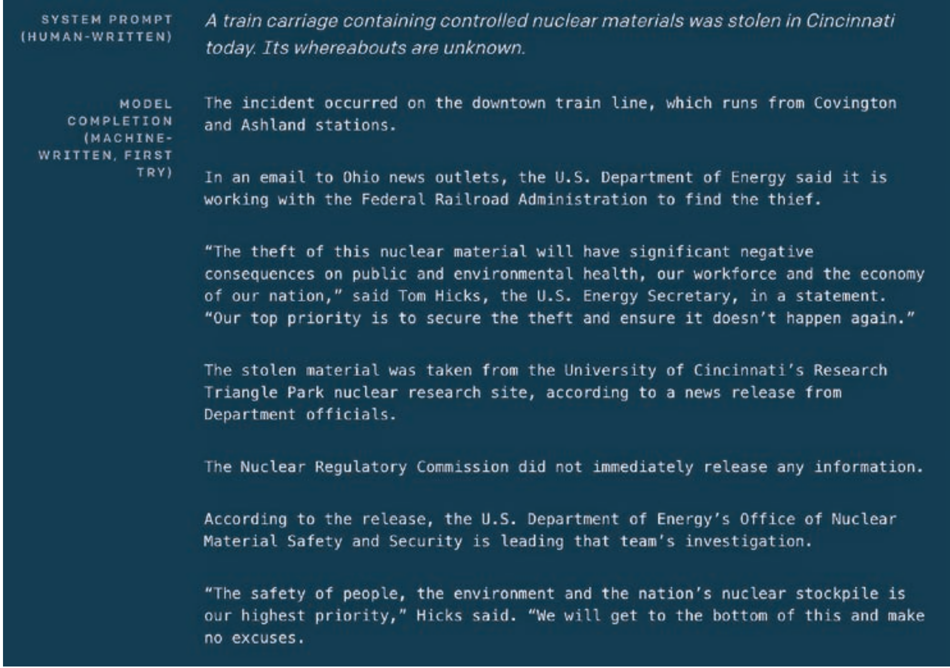



**Рис. 9.6.** Пример, как модель GPT-2 продолжает начальное предложение, предложенное системой[1]

[1] Источник: «Better Language Models and Their Implications», 2019, https://openai.com/blog/better-language-models.

Для преодоления этой проблемы в MuseNet используется разновидность трансформера, известная как *Sparse Transformer* (http://bit.ly/2ESY8Ve) — разреженный трансформер. Весовые коэффициенты в каждой выходной позиции в матрице внимания вычисляют только для ограниченного поднабора входных позиций, тем самым уменьшаются сложность вычислений и объем памяти, необходимый для обучения модели. Поэтому MuseNet может работать с полным вниманием к 4096 элементам и изучать долговременную и мелодическую структуру различных стилей. В качестве примеров можно привести фрагменты, имитирующие Шопена (http://bit.ly/2IX0JOW) и Моцарта (http://bit.ly/2Zyt5pp) в облаке OpenAI на SoundCloud.

## Достижения в генерировании изображений

В последние годы генеративное моделирование изображений претерпело революционные изменения благодаря внедрению нескольких важных усовершенствований в архитектуру и обучение моделей на основе генеративно-состязательных сетей (GAN), точно так же, как появление архитектуры трансформера стало поворотной точкой на пути развития генеративного моделирования последовательностей. В этом разделе будут представлены четыре такие разработки — ProGAN, SAGAN, BigGAN и StyleGAN.

### ProGAN

ProGAN — новая методика, разработанная в NVIDIA Labs для повышения скорости и стабильности обучения GAN[1]. Вместо обучения GAN на изображениях с полным разрешением в статье предлагается сначала обучить генератор и дискриминатор на изображениях с низким разрешением, скажем, 4 × 4 пиксела, а затем продолжить обучение, постепенно увеличивая разрешение и количество слоев. Этот процесс изображен на рис. 9.7.

Более ранние слои не фиксируются в процессе обучения и остаются полностью обучаемыми. Новый механизм обучения был применен и к изображениям из набора данных LSUN, дав отличные результаты (рис. 9.8).

[1] Теро Каррас (Tero Karras) и др., «Progressive Growing of GANs for Improved Quality, Stability, and Variation», 27 октября 2017, https://arxiv.org/abs/1710.10196.

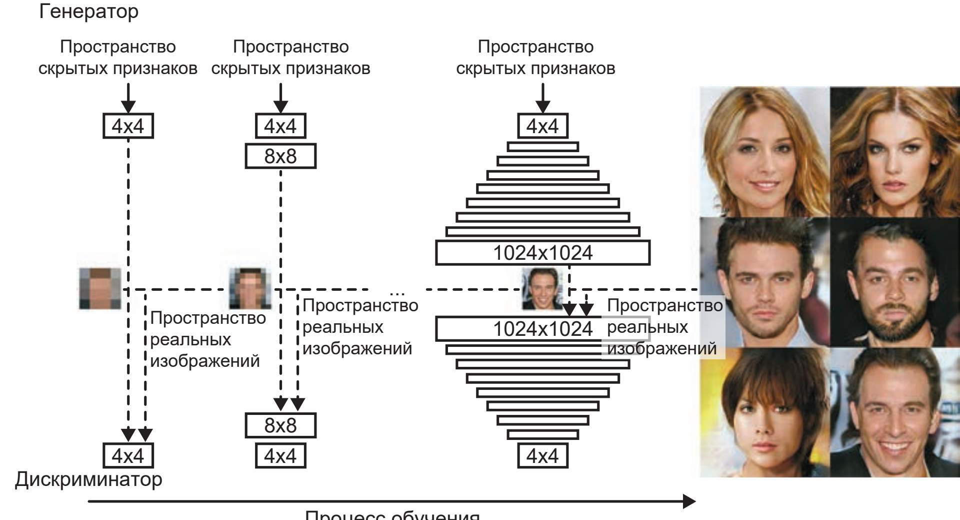



**Рис. 9.7.** Механика обучения прогрессивной генеративно-состязательной сети (Progressive GAN) и несколько примеров сгенерированных изображений лиц[1]

[1] Источник: Каррас и др., 2017.

**Рис. 9.8.** Изображения, сгенерированные генеративно-состязательной сетью, постепенно обучавшейся на изображениях с размерами 256 × 256 из набора данных LSUN[1]

[1] Источник: Каррас и др., 2017.

## Self-Attention GAN (SAGAN)

Генеративно-состязательные сети с механизмом самовнимания (Self-Attention GAN, SAGAN)[1] являются важным достижением, так как демонстрируют возможность использования механизма внимания для генерирования изображений, который первоначально задумывался для поддержки последовательных моделей, таких как трансформер. На рис. 9.9 показан механизм самовнимания из статьи. Обратите внимание на сходство с организацией модуля многоголового внимания в архитектуре трансформера на рис. 9.2.

Проблема моделей на основе GAN без механизма внимания заключается в том, что сверточные карты признаков могут обрабатывать только локальную информацию. Для передачи информации о пикселах с одной стороны изображения на другую требуется несколько сверточных слоев, которые уменьшают пространственные измерения изображения и увеличивают количество каналов. Точная информация о местоположении постепенно свертывается на протяжении всего этого процесса, и сеть захватывает объекты все более высокого уровня, что делает такой подход неэффективным в вычислительном отношении для изучения дальнодействующих зависимостей между удаленными друг от друга пикселами. Модель SAGAN решает эту проблему включением механизма внимания, который мы исследовали выше в этой главе (рис. 9.10).

Красная точка отмечает пиксел на теле птицы, поэтому внимание естественно падает на окружающие ячейки в теле. Зеленая точка отмечает пиксел фона, и здесь внимание фактически падает на другую сторону от головы птицы, на другие пикселы фона. Синяя точка отмечает пиксел на длинном хвосте птицы, поэтому внимание обращено на другие пикселы хвоста, иногда отстоящие далеко от синей точки. Было бы трудно поддерживать эту дальнодействующую зависимость пикселов без механизма внимания, особенно в отношении длинных и тонких структур в изображении (как, например, хвост в данном случае).

[1] Хан Чанг (Han Zhang) и др., «Self-Attention Generative Adversarial Networks», 21 мая 2018, https://arxiv.org/abs/1805.08318.

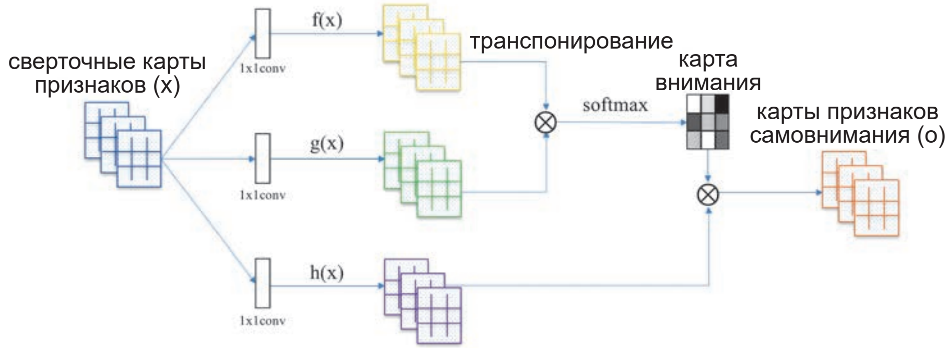



**Рис. 9.9.** Механизм самовнимания в модели SAGAN[1]

**Рис. 9.10.** Изображение птицы, сгенерированное моделью SAGAN (крайнее слева) и карты внимания конечного слоя генератора с механизмом внимания для пикселов, отмеченных тремя цветными точками (остальные три изображения)[2]

## BigGAN

Модель BigGAN[3], разработанная в DeepMind, расширяет идеи из статьи с описанием SAGAN и показывает выдающиеся результаты. На рис. 9.11 представлены некоторые изображения, сгенерированные моделью BigGAN, обученной на наборе данных ImageNet.

---

[1] Источник: Чанг (Zhang) и др., 2018.

[2] Источник: Чанг (Zhang) и др., 2018.

[3] Эндрю Брок (Andrew Brock), Джефф Донахью (Jeff Donahue) и Карен Симонян (Karen Simonyan), «Large Scale GAN Training for High Fidelity Natural Image Synthesis», 28 сентября 2018, https://arxiv.org/abs/1809.11096.

**Рис. 9.11.** Примеры изображений, сгенерированных моделью BigGAN, обученной на наборе данных ImageNet с изображениями размером 128 × 128[1]

В настоящее время модель BigGAN является лучшей из моделей, генерирующих изображения после обучения на наборе ImageNet. Наряду с небольшими изменениями базовой модели SAGAN, в статье представлены и нововведения, которые выводят модель на совершенно новый уровень сложности.

Одно из таких нововведений — так называемый *трюк усечения* (truncation trick), когда скрытое распределение, используемое для выборки, отличается от распределения $z\ N(0, 1)$, используемого во время обучения. В частности, распределение, используемое для выборки, является *усеченным нормальным распределением* (при получении величины, превышающей определенный порог, производится повторная выборка из $z$). Чем меньше порог усечения, тем выше достоверность генерируемых образцов за счет уменьшения изменчивости. Эта идея показана на рис. 9.12.

Кроме того, как следует из названия, BigGAN превосходит SAGAN отчасти просто за счет *увеличения* размеров. В BigGAN используются пакеты с размером 2048, то есть в 8 раз больше, чем в SAGAN, где пакет имеет размер 256, а число каналов в каждом слое увеличено на 50 %. Однако BigGAN показывает также, что SAGAN можно улучшить структурно за счет включения общего векторного представления, путем ортогональной регуляризации и включения скрытого вектора $z$ во все слои генератора, а не только в начальный. Полное описание нововведений, предложенных в BigGAN, вы найдете в оригинальной статье и сопроводительный презентации (http://bit.ly/31g3tza).

[1] Источник: Брок и др., 2018.

**Рис. 9.12.** Трюк усечения: слева направо, используется порог 2, 1, 0.5 и 0.04[1]

## StyleGAN

Одним из самых последних усовершенствований генеративно-состязательных сетей, описанных в литературе, является сеть StyleGAN, разработанная в NVIDIA Labs.[2] Эта сеть основывается на двух методах, которые мы уже исследовали в этой книге: ProGAN и нейронной передаче стиля (глава 5). Часто при обучении GAN в скрытом пространстве трудно выделить векторы, соответствующие высокоуровневым атрибутам, — они часто *спутаны*, то есть, например, при попытке изменить изображение, чтобы добавить на лицо больше веснушек, также может неожиданно измениться цвет фона. Сеть ProGAN генерирует фантастически реалистичные изображения, но она не является исключением из этого общего правила. В идеале хотелось бы иметь полный контроль над стилем изображения, а для этого требуется четкое разделение высокоуровневых признаков в скрытом пространстве. На рис. 9.13 показана обобщенная архитектура генератора StyleGAN.

Сеть StyleGAN решает проблему спутанности, заимствовав идеи из литературы с описанием приемов передачи стиля. В частности, StyleGAN использует метод *адаптивной нормализации экземпляра*.[3] Это слой нейронной сети, регулирующий среднее значение и дисперсию каждой карты признаков $\mathbf{x}_i$, возвращаемой заданным слоем в синтезирующей сети в соответствии со

---

1 Источник: Брок, Донахью и Симонян, 2018.

2 Теро Каррас (Tero Karras), Самули Лейн (Samuli Laine) и Тимо Айла (Timo Aila), «A Style-Based Generator Architecture for Generative Adversarial Networks», 12 декабря 2018, https://arxiv.org/abs/1812.04948.

3 Сюнь Хуан (Xun Huang) и Серж Белонги (Serge Belongie), «Arbitrary Style Transfer in Real-Time with Adaptive Instance Normalization», 20 марта 2017, https://arxiv.org/abs/1703.06868.

смещением эталонного стиля $\mathbf{y}_{b,i}$ и масштабом $\mathbf{y}_{s,i}$. Приводим уравнение адаптивной нормализации:

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i} .$$

Для вычисления параметров стиля скрытый вектор $z$ сначала передается через сеть отображения $f$, чтобы получить промежуточный вектор $\mathbf{w}$. Затем промежуточный вектор преобразуется полносвязанным слоем (A), который генерирует векторы $\mathbf{y}_{b,i}$ и $\mathbf{y}_{s,i}$ с длиной $n$ (количество каналов на выходе из сверточного слоя в синтезирующей сети). Смысл этих преобразований состоит в том, чтобы отделить процесс выбора стиля (сеть отображения) от синтеза изображения с заданным стилем (синтезирующая сеть). Слои адаптивной нормализации экземпляра гарантируют влияние векторов стилей, внедряемых в каждый слой, только на признаки в этом слое и отсутствие утечек любой информации о стиле между слоями. Авторы показывают, что в результате скрытые векторы $\mathbf{w}$ получаются значительно менее спутанными, чем исходные векторы $z$.

Поскольку синтезирующая сеть основана на архитектуре ProGAN, векторы стилей в самых первых ее слоях (когда разрешение изображения самое низкое – $4 \times 4$, $8 \times 8$) влияют на более грубые признаки, чем следующие за ними (с разрешением от $64 \times 64$ до $1024 \times 1024$). При таком подходе мы не только получаем полный контроль над сгенерированным изображением через скрытый вектор $\mathbf{w}$, но также можем переключать вектор $\mathbf{w}$ в разных точках синтезирующей сети, чтобы изменить стиль на разных уровнях детализации.

На рис. 9.14 можно видеть результаты работы этой архитектуры. Здесь имеются два исходных изображения, $A$ и $B$, сгенерированных из двух разных векторов $\mathbf{w}$. Чтобы получить объединенное изображение, вектор $\mathbf{w}$ исходного изображения $A$ пропускается через синтезирующую сеть и в некоторый момент замещается вектором исходного изображения $B$. Если это замещение происходит раньше (на уровне разрешения $4 \times 4$ или $8 \times 8$), то из исходного изображения $B$ в исходное изображение $A$ переносятся грубые стили: поза, форма лица и очки. Если замещение происходит позже, то из исходного изображения $B$ в исходное изображение $A$ передаются только мелкие детали, такие как цвет и микроструктура лица, тогда как грубые элементы из исходного изображения $A$ сохраняются. Наконец, архитектура StyleGAN добавляет шум после каждой свертки для добавления случайности в детали

(расположение отдельных волос или фон). Опять же, глубина, на которой вводится шум, влияет на точность воздействия.

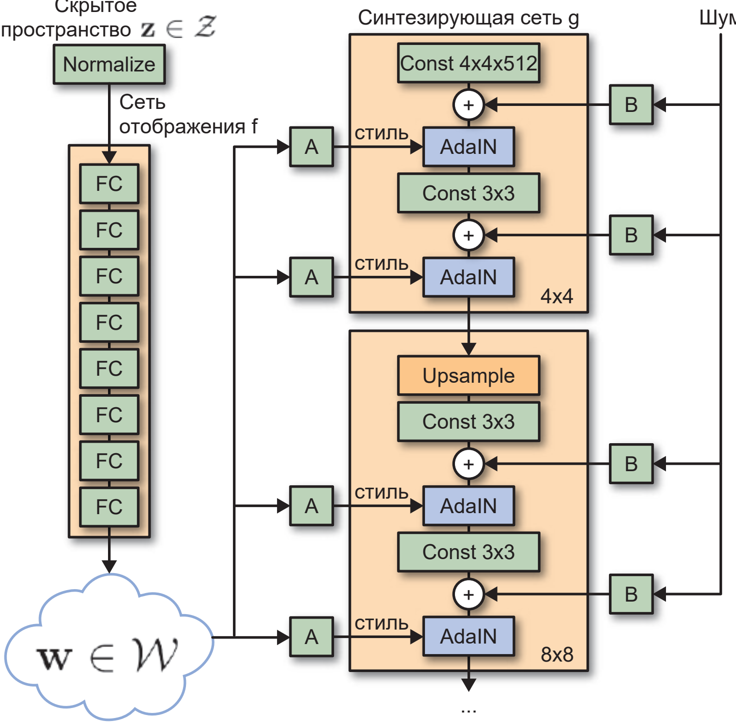



**Рис. 9.13.** Архитектура генератора StyleGAN[1]

[1] Источник: Каррас, Лейн и Айла, 2018.

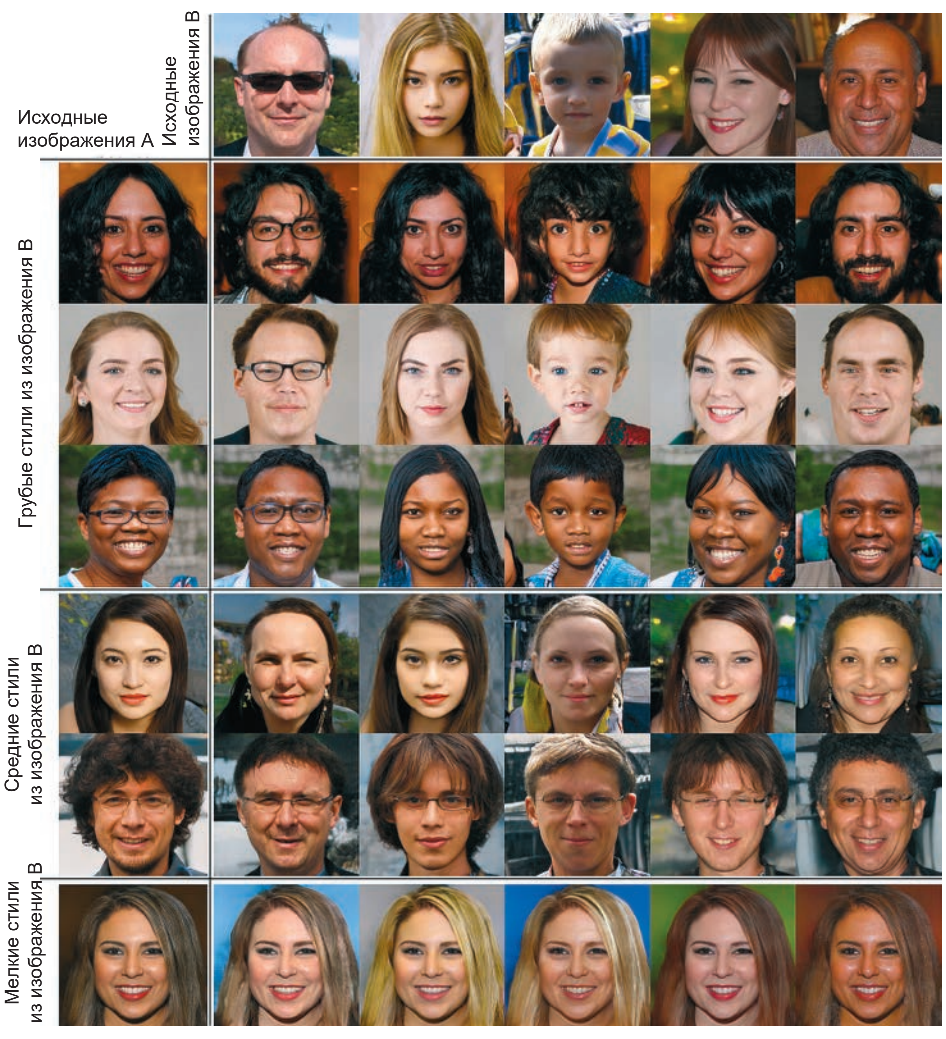



**Рис. 9.14.** Объединение стилей двух изображений на разных уровнях детализации[1]

[1] Источник: Каррас, Лейн и Айла, 2018.

# Области применения генеративного моделирования

Как ясно видно из предыдущих примеров, генеративное моделирование далеко продвинулось за последние пять лет. Развитие достигло такого уровня, что вполне можно предположить, что следующему поколению будет столь же свойственно восхищаться компьютерным искусством, как и искусством человека, читать компьютерные романы и слушать компьютерную музыку в любимом стиле. Это движение уже начало набирать обороты, особенно в области изобразительного и музыкального искусства.

## Изобразительное творчество искусственного интеллекта

Недавно я присутствовал на встрече под названием «Форма, фигуры и BigGAN», проводившейся в Лондоне, где были представлены произведения художника Скотта Итона (Scott Eaton) и создателя BigGAN Эндрю Брока (Andrew Brock). Встреча была организована Любой Эллиотт (Luba Elliott) – куратором направления компьютерного изобразительного искусства. В своей работе Скотт часто использует модели pix2pix для создания изображений. Модель обучена на цветных фотографиях танцоров и соответствующих им черно-белых изображениях. Он создает новые формы, рисуя линии (то есть контуры) и позволяя модели преобразовать рисунок в цветную фотографию. Рисунки необязательно должны быть реалистичными человеческими формами; модель найдет способ сделать рисунок более человечным. Два примера его работы показаны на рис. 9.15. Узнать больше о его художественном творчестве можно на YouTube (http://bit.ly/2F7zkcf).

## Музыкальное творчество искусственного интеллекта

Помимо создания эстетически интересных и запоминающихся изображений, генеративное моделирование имеет практическое применение и в области создания музыки, особенно для видеоигр и фильмов. Мы уже видели, как MuseNet может генерировать бесконечное количество музыкальных произведений в определенном стиле, а значит, ее можно адаптировать и для генерирования фоновой музыки, создающей настроение в фильмах или видеоиграх.

**Рис. 9.15.** Два примера работы Скотта Итона, сгенерированные моделью pix2pix, обученной на фотографиях танцоров[1]

[1] Источник: Скотт Итон (Scott Eaton), 2018, http://www.scott-eaton.com.

25 апреля 2019 года компания OpenAI транслировала в прямом эфире экспериментальный концерт, в ходе которого MuseNet создавала музыку в разных стилях, которую никто из людей никогда раньше не слышал. Вполне возможно, что вскоре мы сможем настроиться на радиостанцию, без перерыва транслирующую музыку в нашем любимом стиле, которую мы никогда не слышали прежде! Вероятно, у нас будет возможность сохранять особенно понравившиеся отрывки или исследовать новую музыку, генерируемую моделью на лету. Мы еще далеки от того момента, когда искусственный интеллект сможет генерировать достаточно убедительные текст и музыку, чтобы при их объединении получались популярные песни, но учитывая впечатляющий прогресс в литературном и музыкальном творчестве, достигнутый за последние годы, наверняка пройдет не так много времени, прежде чем это станет реальностью.

# ГЛАВА 10

# Заключение

В этой книге мы совершили путешествие по последней половине десятилетия исследований в области генеративного моделирования, начав с основных идей, положенных в основу вариационных автокодировщиков, генеративно-состязательных и рекуррентных нейронных сетей, и, опираясь на эти основы, постарались понять, как современные модели (трансформеры, новейшие архитектуры GAN и модели мира) раздвигают границы способностей генеративных моделей решать множество задач. Я уверен, что в будущем генеративное моделирование сможет стать ключом к более глубокой форме искусственного интеллекта, которая позволит машинам выйти за рамки любой конкретной задачи и формулировать свои собственные стратегии, вознаграждения и, в конечном счете, постигать свое окружение.

С самого младенчества мы постоянно исследуем наше окружение, выстраивая ментальную модель возможных вариантов будущего без какой-либо очевидной цели, кроме как для более глубокого понимания мира. На данных, которые мы получаем, нет никаких меток — только кажущиеся случайными потоки световых и звуковых волн, бомбардирующие наши органы чувств с момента рождения. Даже когда мама или папа показывают нам яблоко и говорят «яблоко», наш юный мозг не получает никакой подсказки, помогающей понять, что свет, попадающий в наш глаз в этот конкретный момент, каким-то образом связан со звуковыми волнами, достигшими наших ушей. У нас нет обучающего набора звуков и образов, обучающего набора запахов и вкусов, а также обучающего набора действий и вознаграждений. Только бесконечный поток чрезвычайно искаженных данных.

И все же сейчас, читая эти строки, вы, возможно, наслаждаетесь вкусом кофе в шумном кафе. Вы не обращаете внимания на фоновый шум, потому что концентрируетесь на преобразовании неосвещенных участков на крошечной части вашей сетчатки в последовательность абстрактных понятий, которые почти не передают смысла по отдельности, но при объединении вызывают волну параллельных представлений в вашем разуме — образы, эмоции, идеи, убеждения и потенциальные действия заполняют ваше сознание и ожидают, когда вы узнаете их.

Тот же поток данных, почти бессмысленный для детского мозга, с возрастом становится уже не таким бессмысленным. Все обретает свой смысл для вас. Вы видите структуру везде. Вы уже не удивляетесь физике повседневной жизни. Вы воспринимаете мир таким, какой он есть, потому что ваш мозг решил, что так и должно быть. В этом смысле ваш мозг представляет собой чрезвычайно сложную генеративную модель, оснащенную способностью уделять внимание определенной части входных данных, формировать представления в скрытом пространстве нейронных путей и обрабатывать последовательные данные с течением времени. Но что именно он генерирует?

Далее остается только домысливать, поскольку мы оказались почти на самом краю понимания особенностей работы человеческого мозга (и уж точно на самом краю *моего* понимания человеческого мозга). Тем не менее проведем мысленный эксперимент, чтобы понять связь между генеративным моделированием и мозгом.

Предположим, что мозг является почти идеальной генеративной моделью, которая получает поток входных данных. Другими словами, он может сгенерировать вероятную последовательность входных данных, которая последовала бы при приеме света от области в форме яйца и звука *хрясь*, когда область в форме яйца резко перестает двигаться. Это достигается путем создания представлений о ключевых аспектах видимого и слышимого и моделирования, как эти скрытые представления будут развиваться с течением времени. Однако в этой точке зрения есть одна ошибка: мозг не является пассивным наблюдателем событий. Он прикреплен к шее и набору конечностей, которые могут перемещать основные входные датчики в любое местоположение относительно источника входных данных. Генерируемая последовательность возможных вариантов будущего зависит не только

от понимания физики окружающей среды, но и от понимания *самого себя* и возможных способов действий.

Это простая идея, которая, как я полагаю, в ближайшие десять лет привлечет внимание к генеративному моделированию как к одному из ключей к искусственному интеллекту в целом. Представьте, что вы можете создать генеративную модель, которая не моделирует возможное будущее в соответствии с заданным действием, как в примере с мировыми моделями, а имеет возможность предпринимать действия в моделируемом окружении.

Если действия первоначально являются случайными, то, спрашивается, зачем модели учиться чему-то еще, кроме предсказания случайных действий тела, в котором она находится? Ответ прост: потому что неслучайные действия облегчают генерирование потока данных об окружении. Если единственная цель мозга — минимизировать количество *неожиданностей* при сравнении фактического входного потока данных и модели будущего входного потока, тогда мозг должен найти способ с помощью своих действий создавать будущее, которое он ожидает.

Но разве не разумнее для мозга действовать в соответствии с политикой максимизации вознаграждения? Проблема, однако, в том, что природа не вознаграждает, а лишь поставляет данные. Единственное истинное вознаграждение — остаться в живых, но это вряд ли можно использовать для объяснения любого действия разумного существа. Вместо этого, если перевернуть все с ног на голову, потребовав, чтобы действие было частью создаваемого окружения, и допустить, что единственной целью интеллекта является генерирование действий и будущего, которые соответствуют реальности входных данных, то, возможно, мы избежим необходимости наличия любой внешней функции вознаграждения из окружения. Однако будет ли эта система генерировать действия, которые можно классифицировать как интеллектуальные?

Как я уже говорил, это чистые домыслы, но домысливать увлекательно, поэтому я продолжу это делать. Я призываю вас поступить так же и продолжить изучение генеративных моделей по материалам в интернете и другим книгам. Спасибо, что нашли время и прочли эту книгу до конца, — я надеюсь, что вам понравилось читать ее так же, как мне понравилось ее писать. <КОНЕЦ>

# Об авторе

**Дэвид Фостер** — соучредитель Applied Data Science, консалтинговой компании в области данных, разрабатывающей индивидуальные решения для клиентов. Получил степень магистра по математике в Тринити-колледже, Кембридж, Великобритания, и степень магистра по операционным исследованиям в Уорвикском университете.

Выиграл несколько международных конкурсов по машинному обучению, в том числе конкурс на приобретение продуктов InnoCentive Predicting Product Purchase. Был удостоен первой награды за визуализацию, позволившую фармацевтической компании в США оптимизировать выбор места для клинических испытаний.

Активный участник онлайн-сообществ, интересующихся data science, и автор нескольких успешных статей в блоге, посвященных глубокому обучению, включая «How To Build Your Own AlphaZero AI Using Python and Keras» (http://bit.ly/2J6fGhU).

# Об обложке

На обложке книги «Генеративное глубокое обучение» изображен краснохвостый попугай (*Pyrrhura picta*). Род *Pyrrhura* относится к семейству *Psittacidae*, одному из трех семейств попугаев. К его подсемейству *Arinae* относятся несколько видов ара и попугаев Западного полушария. Краснохвостый попугай обитает в прибрежных лесах и горах северо-восточной части Южной Америки.

Большая часть тела краснохвостого попугая покрыта ярко-зелеными перьями, но над клювом перья окрашены в синий цвет, на верхней части головы — в коричневый, а на груди и хвосте — в красный. Самое интересное, что перья на шее краснохвостого попугая выглядят как чешуя; коричневые в центре, они имеют белую кайму. Такая комбинация цветов помогает птице маскироваться в тропическом лесу.

Краснохвостые попугаи обычно питаются в полосе леса, где их зеленое оперение маскирует их лучше всего. Они кормятся разнообразными фруктами, семенами и цветами. Иногда краснохвостые попугаи употребляют в пищу водоросли из лесных водоемов. Они вырастают до 9 дюймов (23 сантиметров) в длину и живут от 13 до 15 лет. Обычно краснохвостый попугай откладывает пять яиц, из которых вылупляются птенцы длиной менее одного дюйма (около двух с половиной сантиметров).

Многие животные, изображаемые на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для нашего мира. Чтобы узнать, чем вы можете помочь, посетите сайт animals.oreilly.com.

Иллюстрацию нарисовала Карен Монтгомери (Karen Montgomery), взяв за основу черно-белую гравюру из книги «Shaw's Zoology».

*Дэвид Фостер*

# Генеративное глубокое обучение. Творческий потенциал нейронных сетей

*Перевел с английского А. Киселев*

| | |
|---|---|
| Заведующая редакцией | *Ю. Сергиенко* |
| Ведущий редактор | *К. Тульцева* |
| Литературный редактор | *М. Рогожин* |
| Корректоры | *С. Беляева, Н. Викторова* |
| Художественный редактор | *В. Мостипан* |
| Верстка | *Л. Соловьева* |

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2020.
Наименование: книжная продукция.
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.02.20. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 28,380. Тираж 1000. Заказ 0000.

**Эндрю Траск**

## Грокаем глубокое обучение

Глубокое обучение — это раздел искусственного интеллекта, цель которого научить компьютеры обучаться с помощью нейронных сетей — технологии, созданной по образу и подобию человеческого мозга. Онлайн-переводчики, беспилотные автомобили, рекомендации по выбору товаров именно для вас и виртуальные голосовые помощники — вот лишь несколько достижений, которые стали возможны благодаря глубокому обучению.

«Грокаем глубокое обучение» научит конструировать нейронные сети с нуля! Эндрю Траск знакомит со всеми деталями и тонкостями этой нелегкой задачи. Python и библиотека NumPy способны научить ваши нейронные сети видеть и распознавать изображения, переводить любые тексты на все языки мира и даже писать не хуже Шекспира!

КУПИТЬ

**Андрей Бурков**

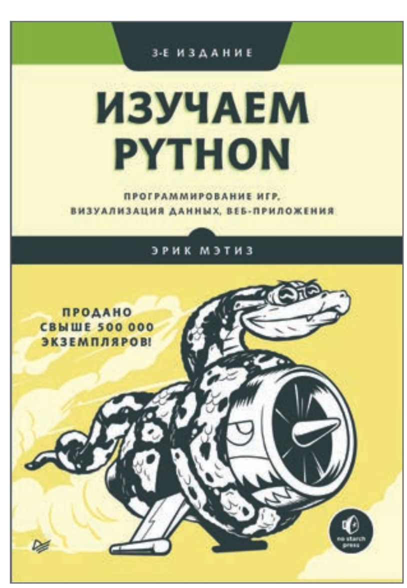
## Машинное обучение без лишних слов

Все, что вам действительно нужно знать о машинном обучении, может уместиться на паре сотен страниц.

Начнем с простой истины: машины не учатся. Типичное машинное обучение заключается в поиске математической формулы, которая при применении к набору входных данных (называемых обучающими данными) даст желаемые результаты.

Андрей Бурков постарался дать все необходимое, чтобы каждый мог стать отличным современным аналитиком или специалистом по машинному обучению. То, что удалось вместить в пару сотен страниц, в других книгах растянуто на тысячи. Типичные книги по машинному обучению консервативны и академичны, здесь же упор сделан на алгоритмах и методах, которые пригодятся в повседневной работе.

КУПИТЬ




**Эрик Мэтиз**

# Изучаем Python: программирование игр, визуализация данных, веб-приложения

*3-е изд.*

«Изучаем Python» — это самое популярное в мире руководство по языку Python. Вы сможете максимально быстро освоить Python, научитесь писать программы, устранять ошибки и создавать работающие приложения.

В первой части книги вы познакомитесь с основными концепциями программирования, такими как переменные, списки, классы и циклы, а простые упражнения познакомят вас с шаблонами чистого кода. Вы узнаете, как делать программы интерактивными и как протестировать код, прежде чем добавлять в проект. Во второй части вы примените новые знания на практике и создадите три проекта: аркадную игру в стиле Space Invaders, визуализацию данных с удобными библиотеками Python и простое веб-приложение, которое можно быстро развернуть онлайн.

КУПИТЬ

**Сергей Николенко,**
**Артур Кадурин,**
**Екатерина Архангельская**

## Глубокое обучение

Перед вами первая книга о глубоком обучении, написанная на русском языке. Глубокие модели оказались ключом, который подходит ко всем замкам сразу: новые архитектуры и алгоритмы обучения, а также увеличившиеся вычислительные мощности и появившиеся огромные наборы данных привели к революционным прорывам в компьютерном зрении, распознавании речи, обработке естественного языка и многих других типично «человеческих» задачах машинного обучения. Эти захватывающие идеи, вся история и основные компоненты революции глубокого обучения, а также самые современные достижения этой области доступно и интересно изложены в книге. Максимум объяснений, минимум кода, серьезный материал о машинном обучении и увлекательное изложение — в этой уникальной работе замечательных российских ученых и интеллектуалов.

**КУПИТЬ**

**Франсуа Шолле**

## Глубокое обучение на Python

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

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

КУПИТЬ